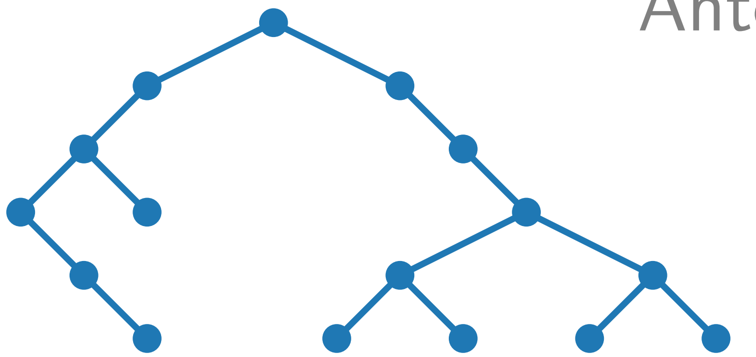
Visualisation of graphs

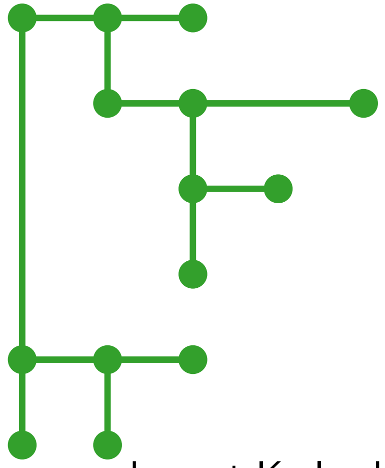
Drawing trees and series-parallel graphs

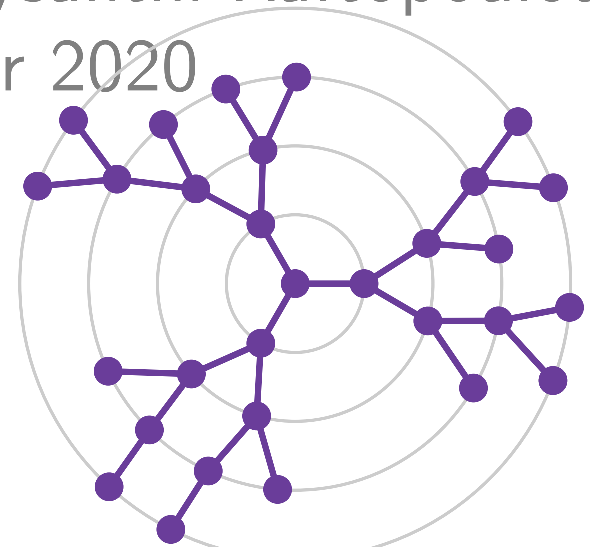
Divide and conquer methods

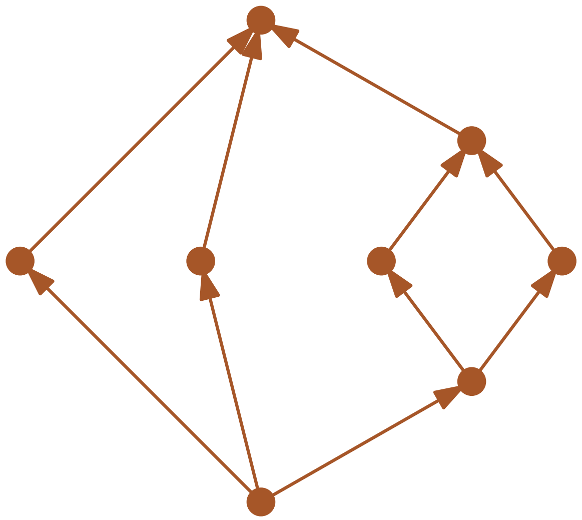
Antonios Symvonis · Chrysanthi Raftopoulou

Fall semester 2020





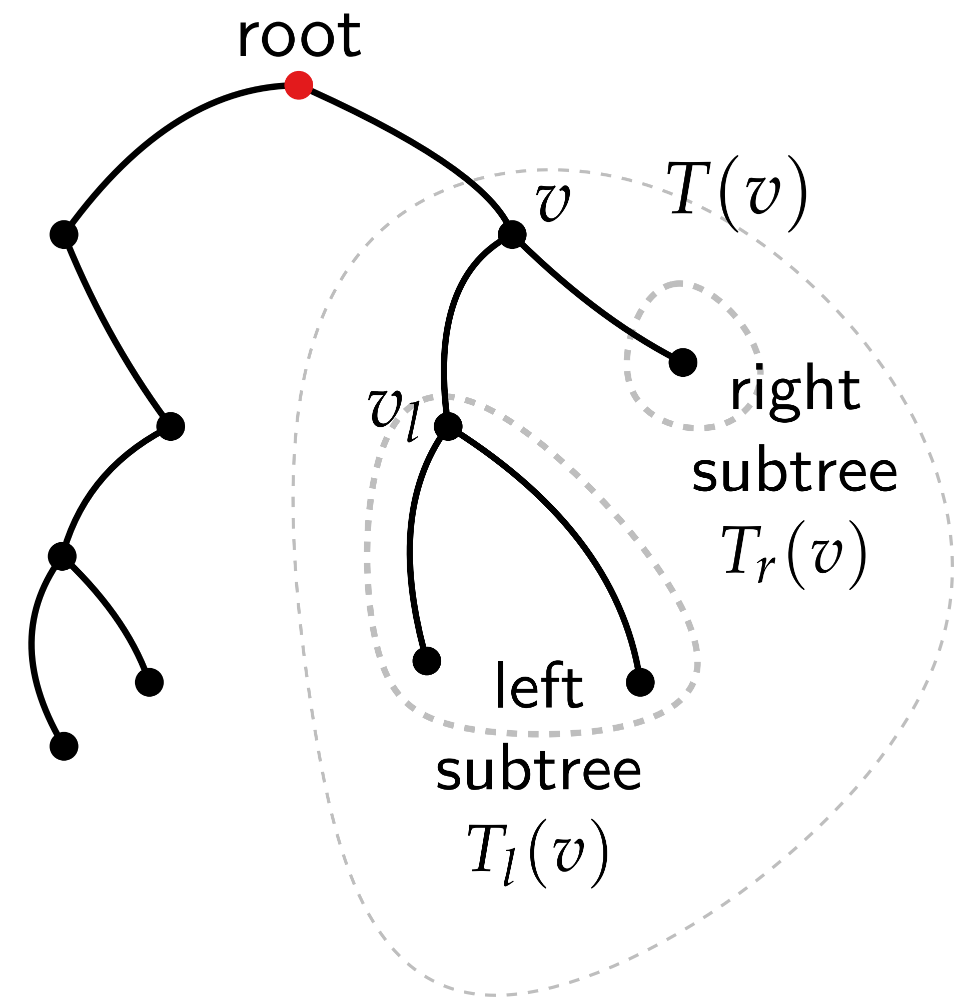




The original slides of this presentation were created by researchers at Karlsruhe Institute of Technology (KIT), TU Wien, U Wuerzburg, U Konstanz, ...
The original presentation was modified/updated by A. Symvonis

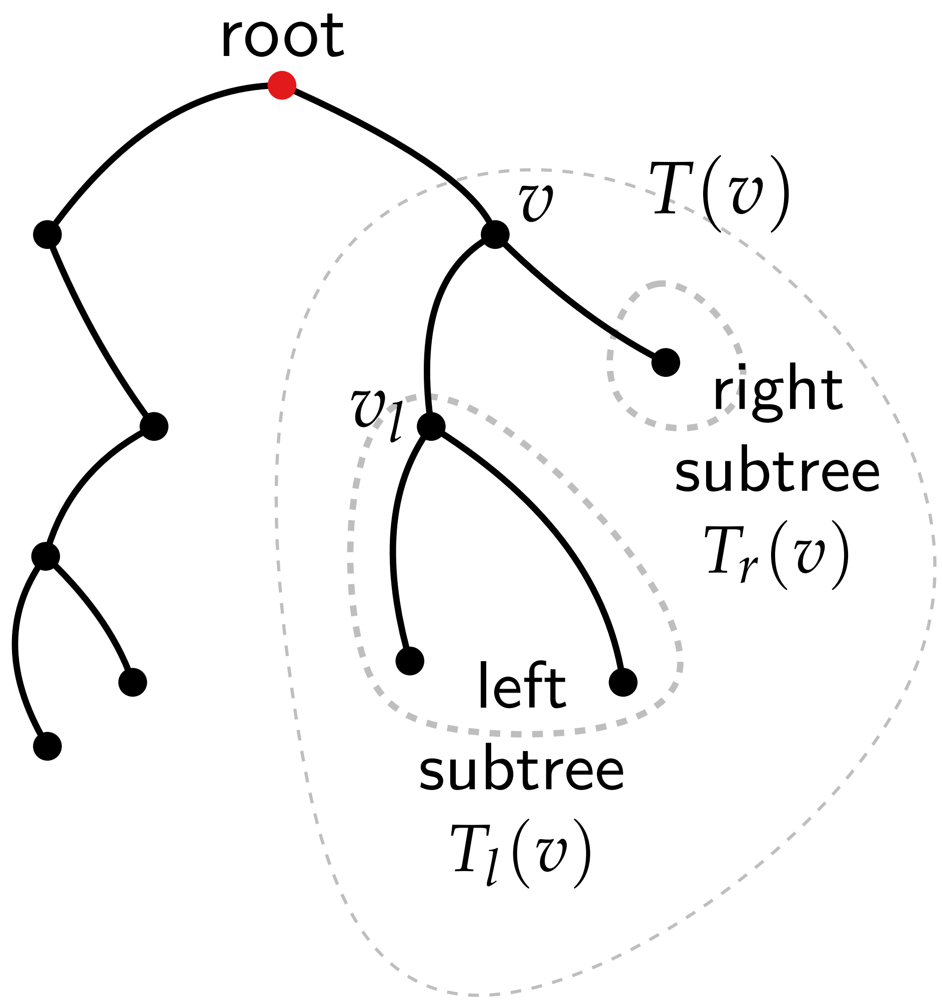
Trees

- Tree - connected graph without cycles
- here: binary and rooted



Trees

- Tree - connected graph without cycles
- here: binary and rooted



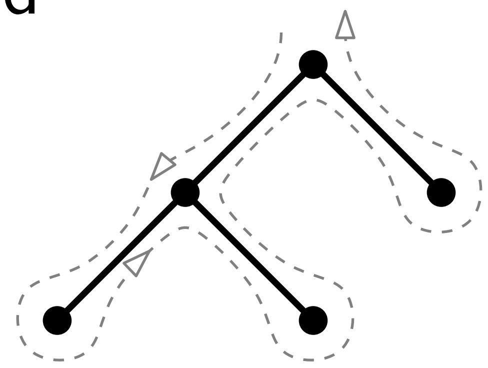
Tree traversal

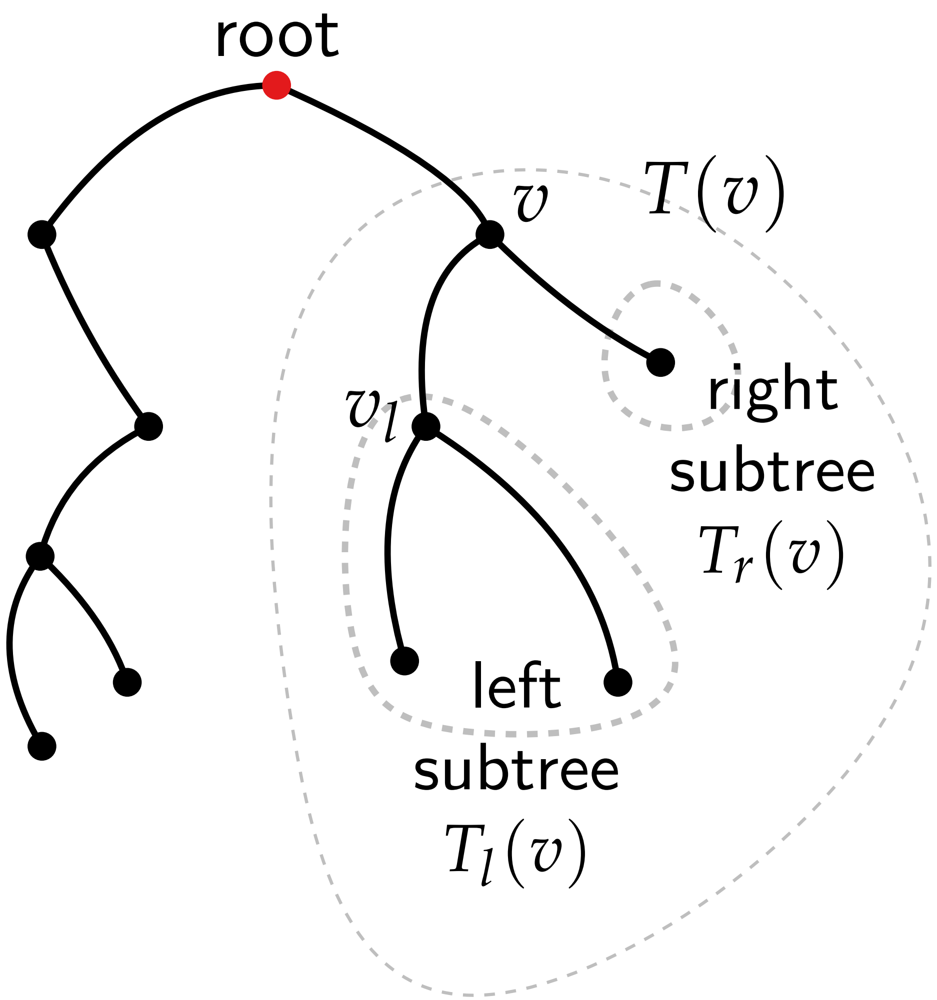
Trees

- Tree - connected graph without cycles
- here: binary and rooted

Tree traversal

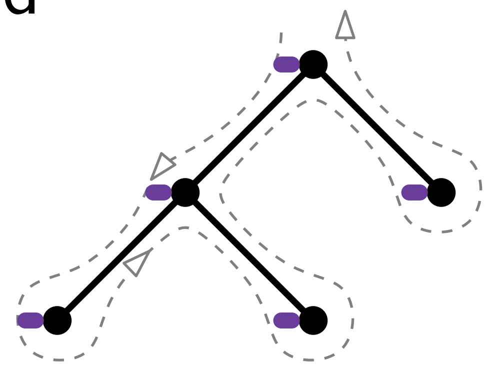
- Depth-first search





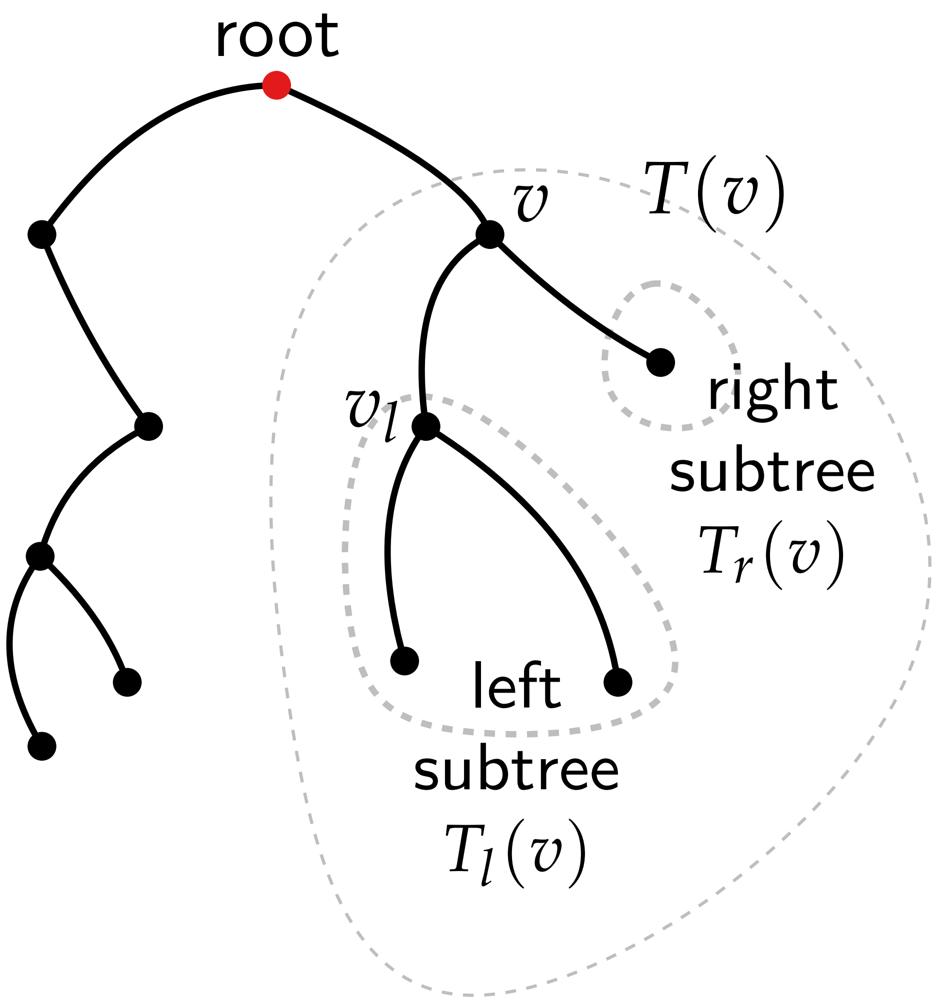
Trees

- Tree - connected graph without cycles
- here: binary and rooted



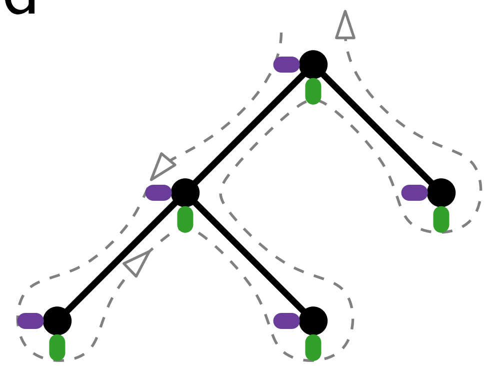
Tree traversal

- Depth-first search
  - Pre-order – first parent, then subtrees



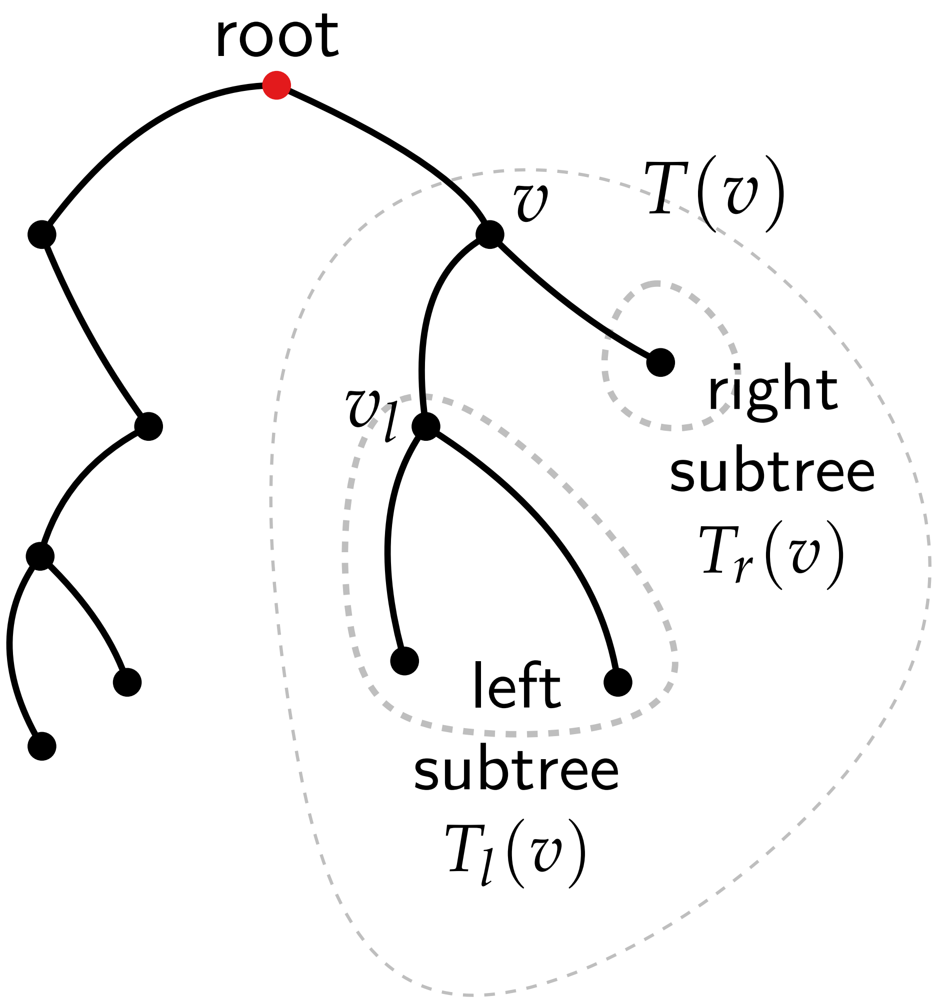
Trees

- Tree - connected graph without cycles
- here: binary and rooted



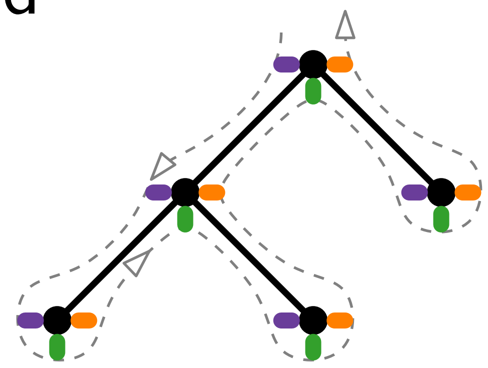
Tree traversal

- Depth-first search
  - Pre-order – first parent, then subtrees
  - In-order – left child, parent, right child



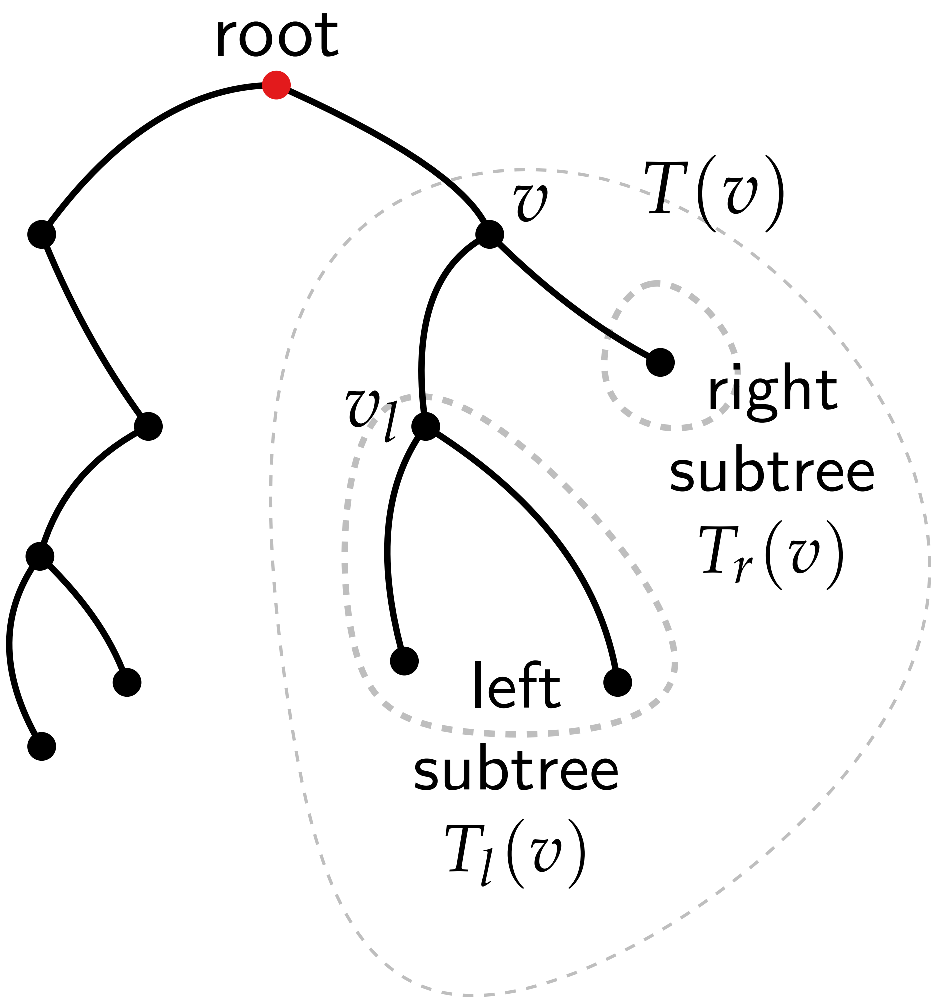
Trees

- Tree - connected graph without cycles
- here: binary and rooted



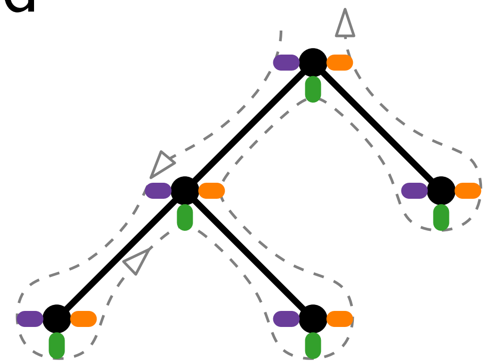
Tree traversal

- Depth-first search
  - Pre-order – first parent, then subtrees
  - In-order – left child, parent, right child
  - Post-order – first subtrees, then parent



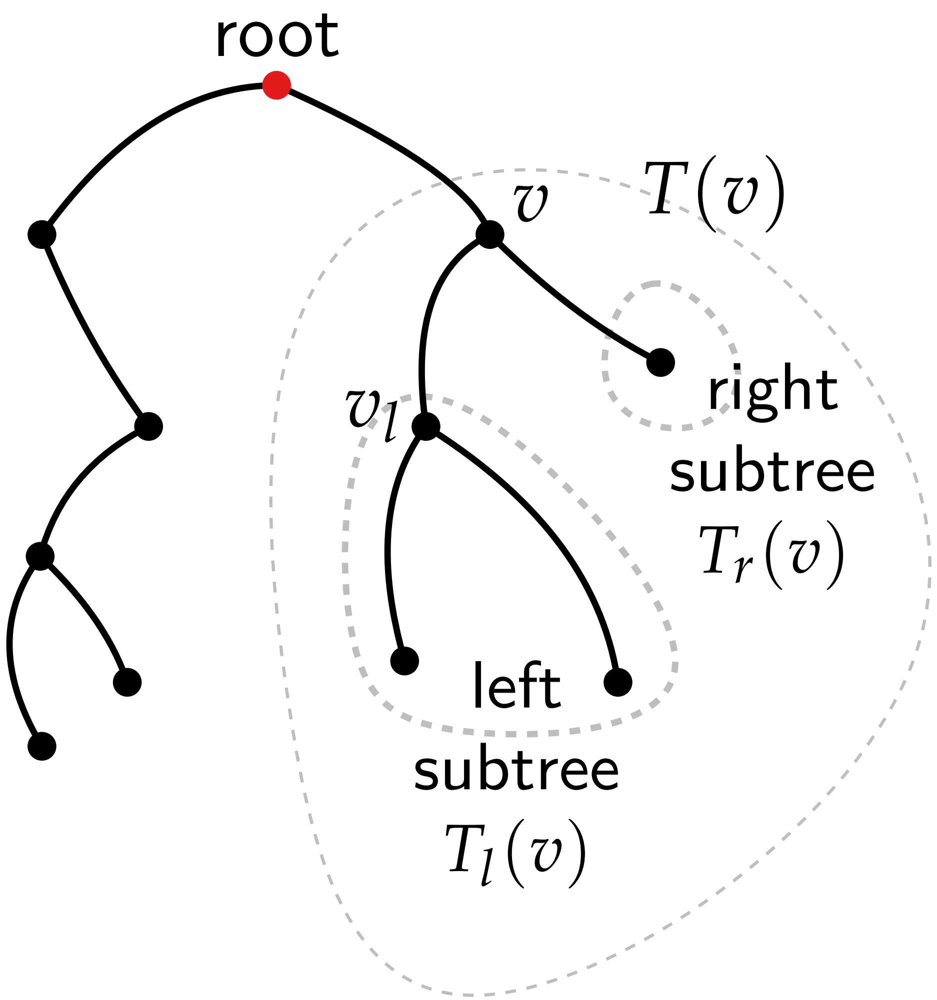
Trees

- Tree - connected graph without cycles
- here: binary and rooted



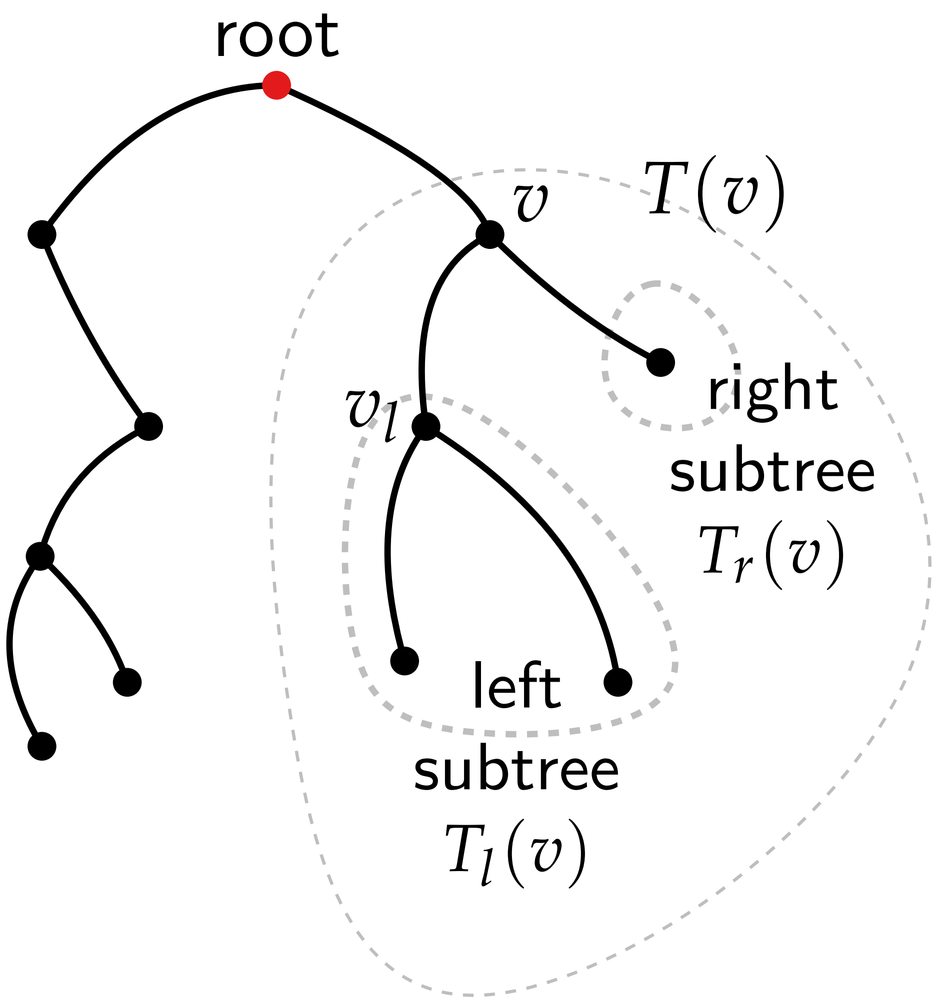
Tree traversal

- Depth-first search
  - Pre-order – first parent, then subtrees
  - In-order – left child, parent, right child
  - Post-order – first subtrees, then parent
- Breadth-first search
  - Assignes vertices to levels corresponding to depth

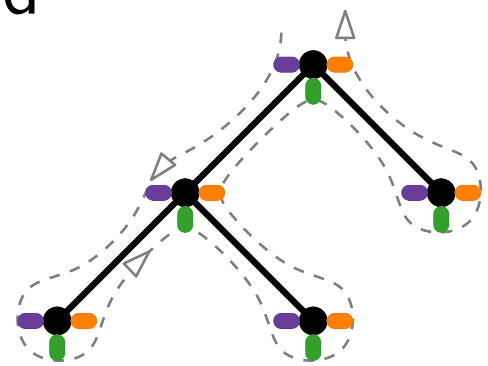


Trees

- Tree - connected graph without cycles
- here: binary and rooted



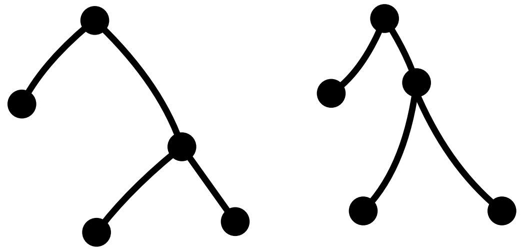
Tree traversal



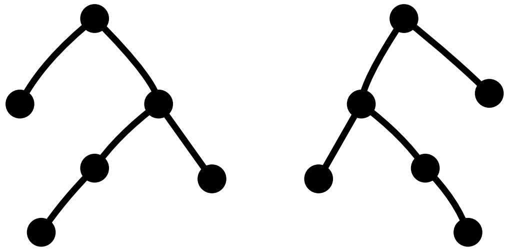
- Depth-first search
  - Pre-order – first parent, then subtrees
  - In-order – left child, parent, right child
  - Post-order – first subtrees, then parent
- Breadth-first search
  - Assignes vertices to levels corresponding to depth

Isomporphism

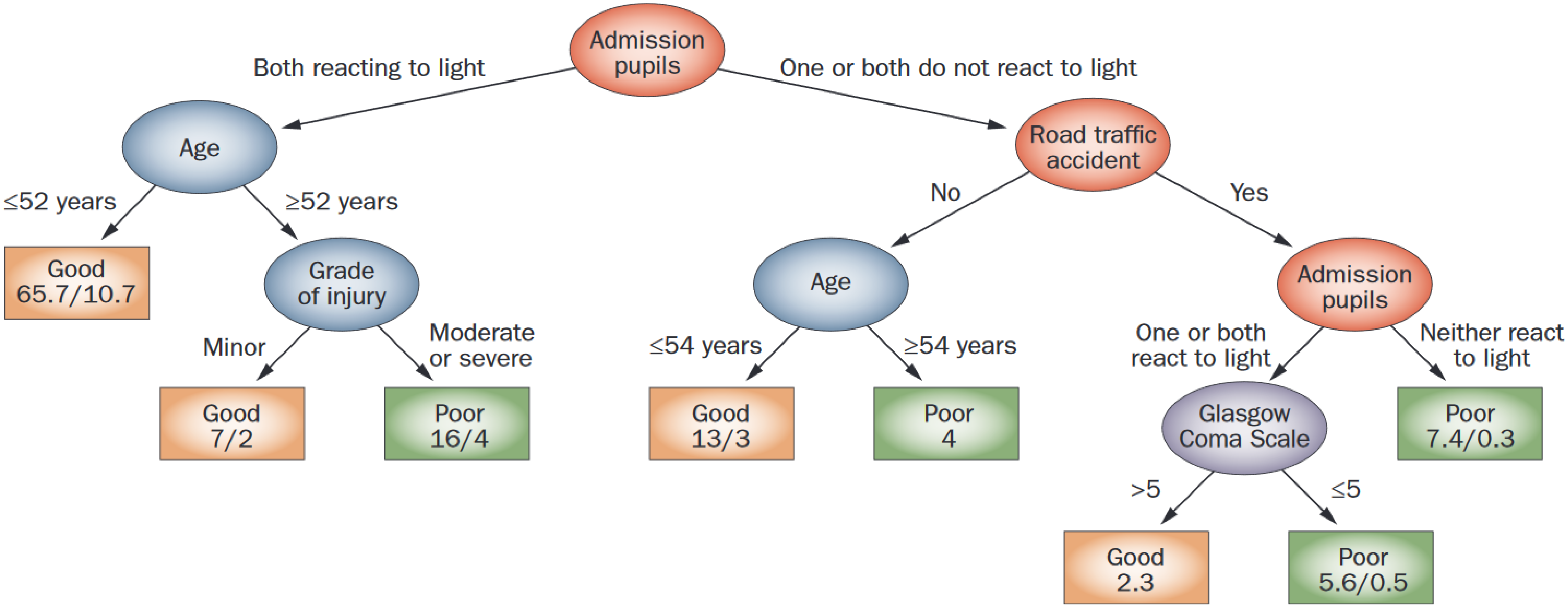
simple



axial



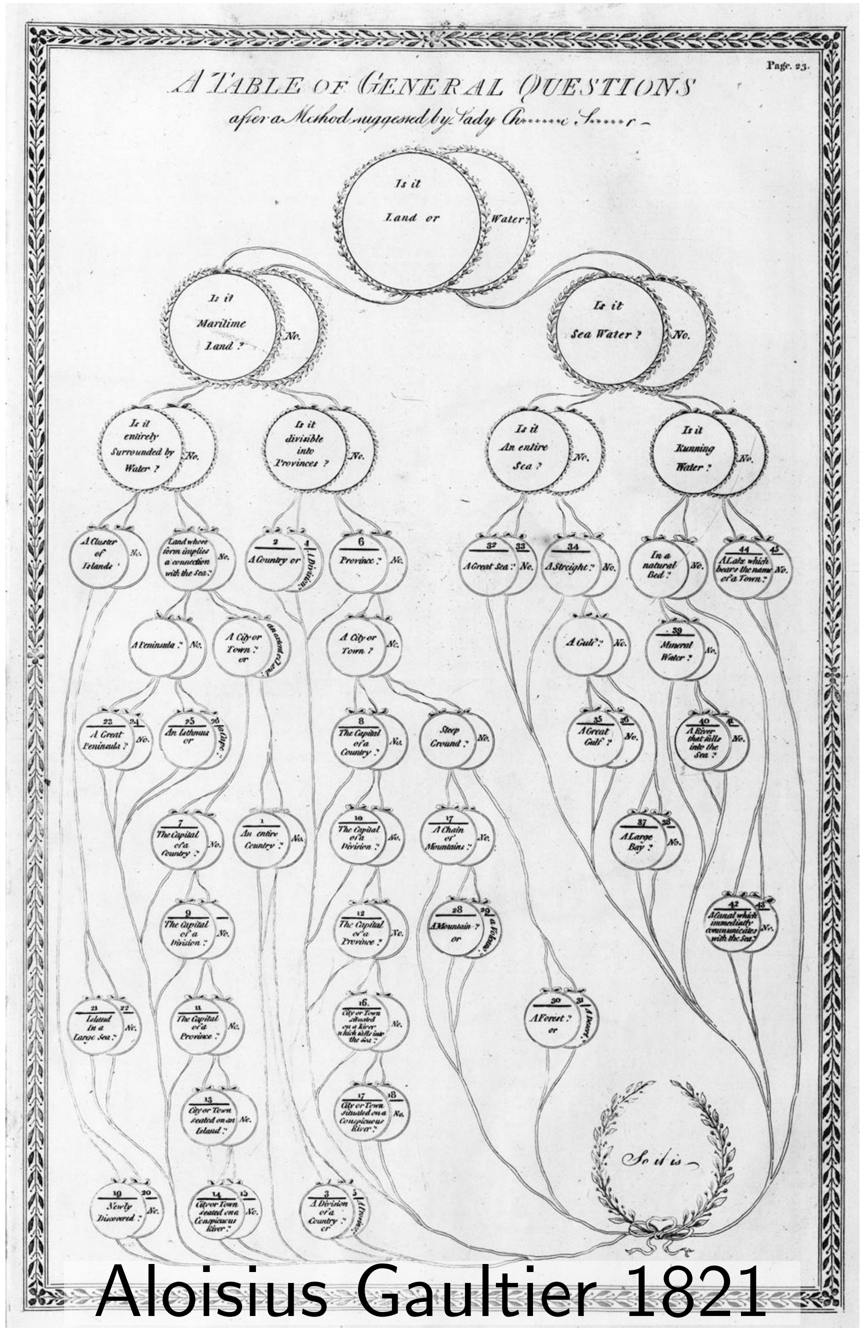
Level-based layout – applications



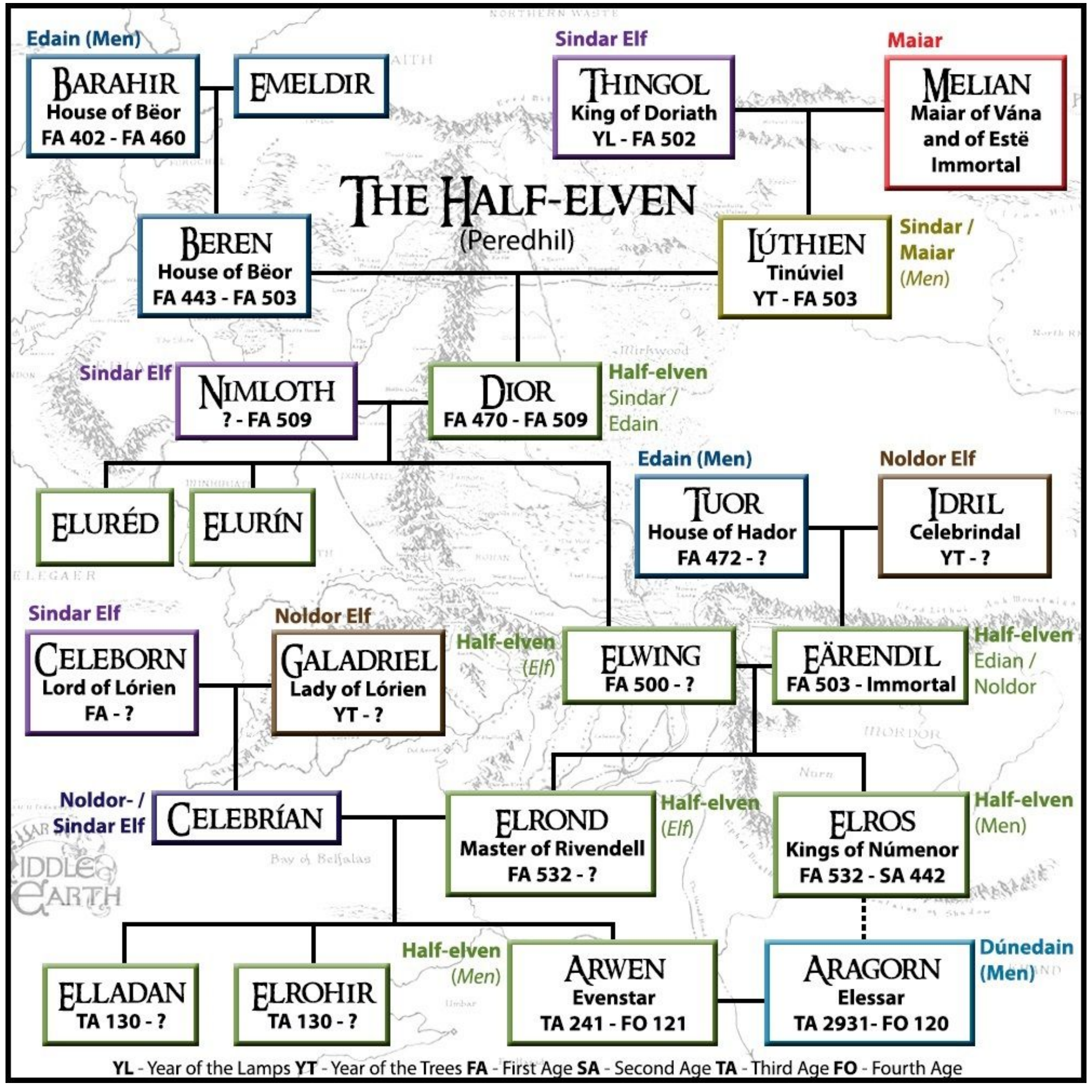
Decision tree for outcome prediction after traumatic brain injury

Source: Nature Reviews Neurology

Level-based layout – applications

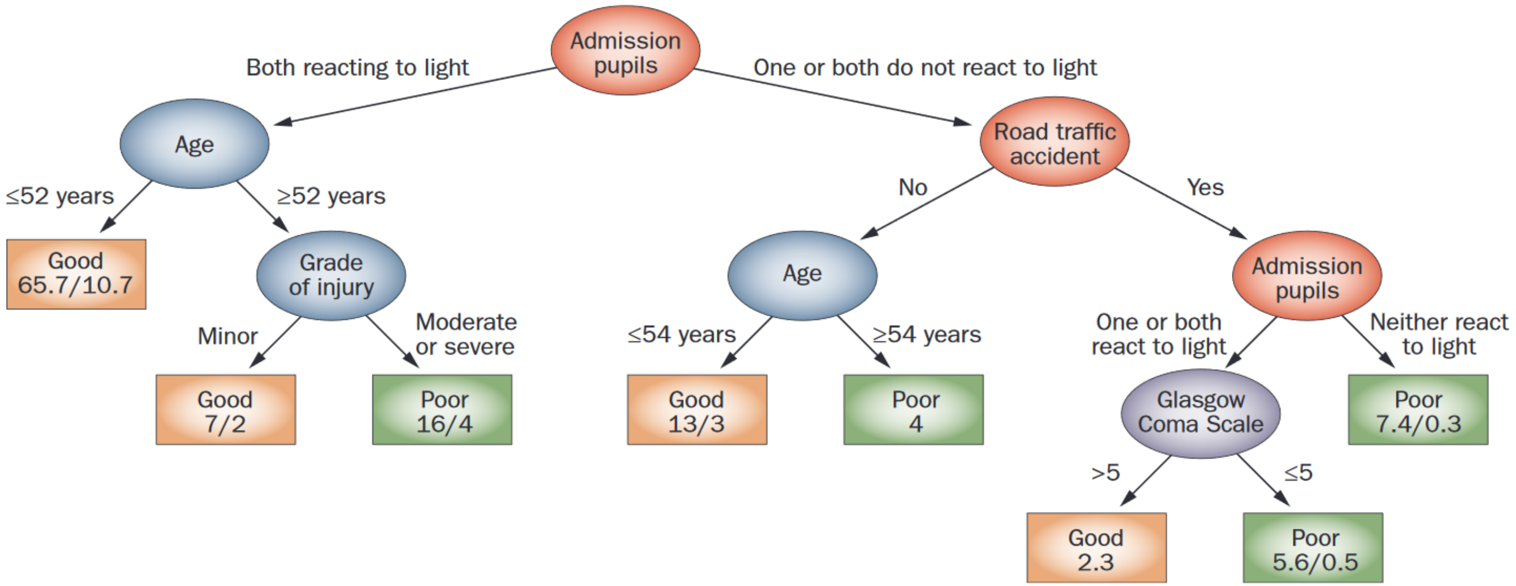


Aloisius Gaultier 1821



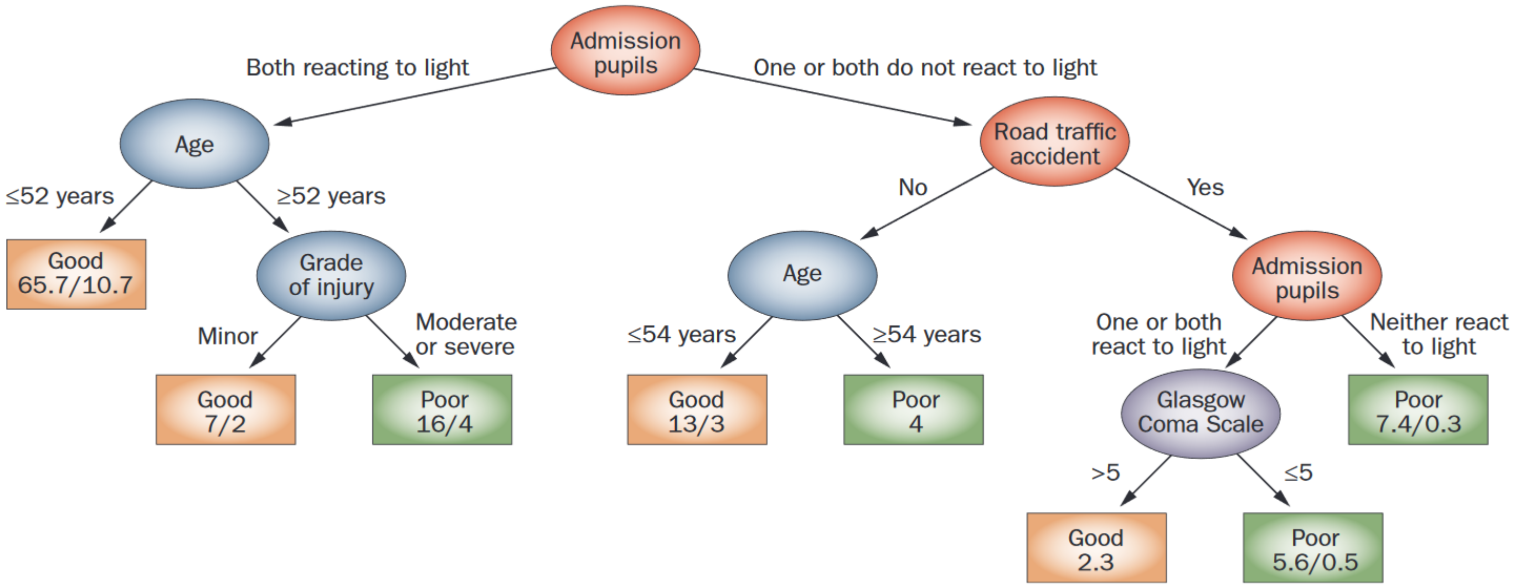
Family tree of LOTR
elves and half-elves

Level-based layout – drawing style



- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimise?

Level-based layout – drawing style

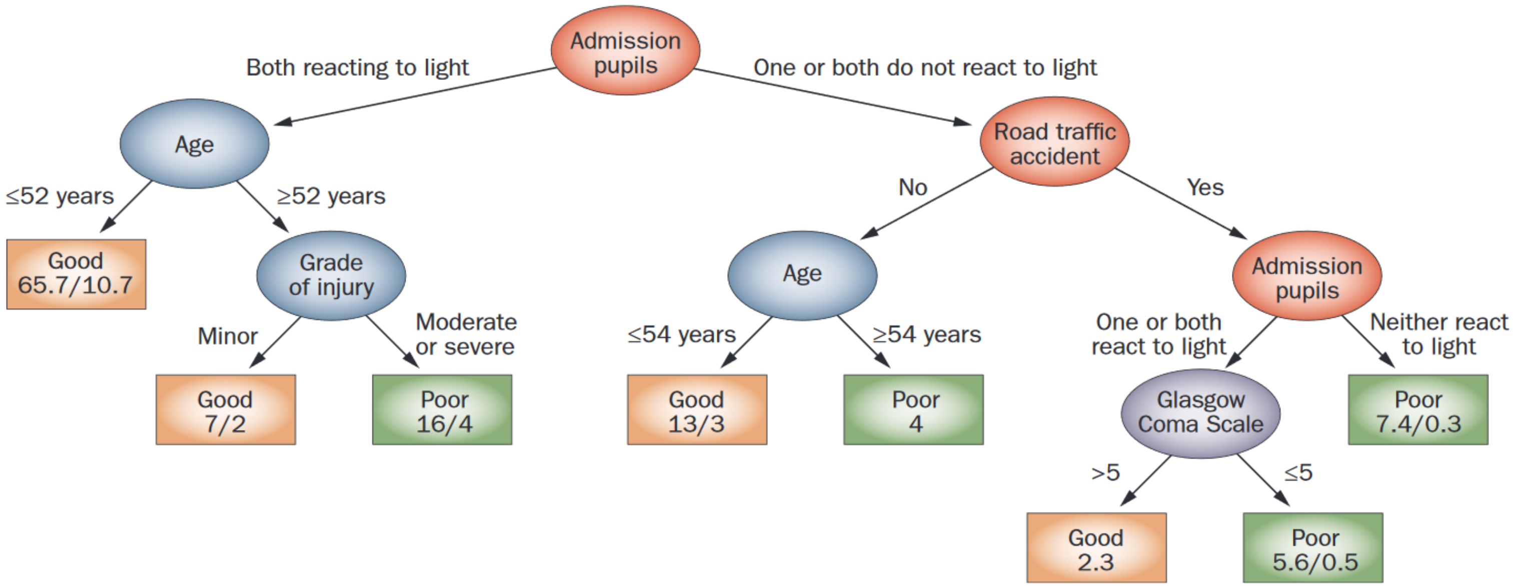


- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimise?

Drawing conventions

- Vertices lie on layers and have integer coordinates
- Parent above children and “within their X-range” (typically, centered)
- Edges are straight-line segments
- Isomorphic subtrees have identical drawings

Level-based layout – drawing style



- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimise?

Drawing conventions

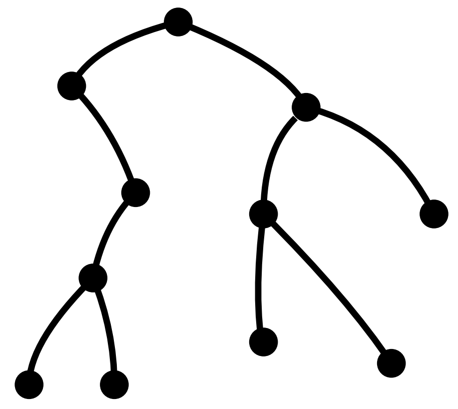
- Vertices lie on layers and have integer coordinates
- Parent above children and “within their X-range” (typically, centered)
- Edges are straight-line segments
- Isomorphic subtrees have identical drawings

Drawing aesthetics

- Area

Level-based layout A simple approach

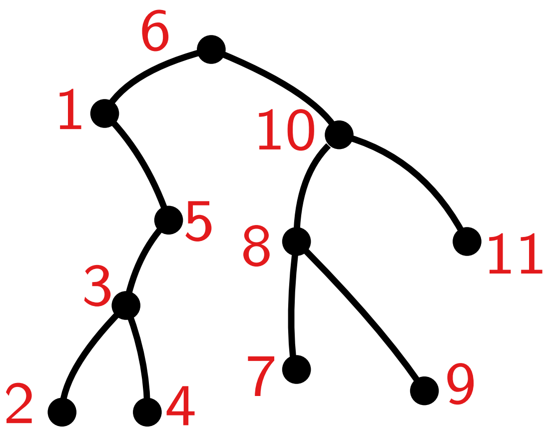
Input: A binary tree T
Output: A leveled drawing of T



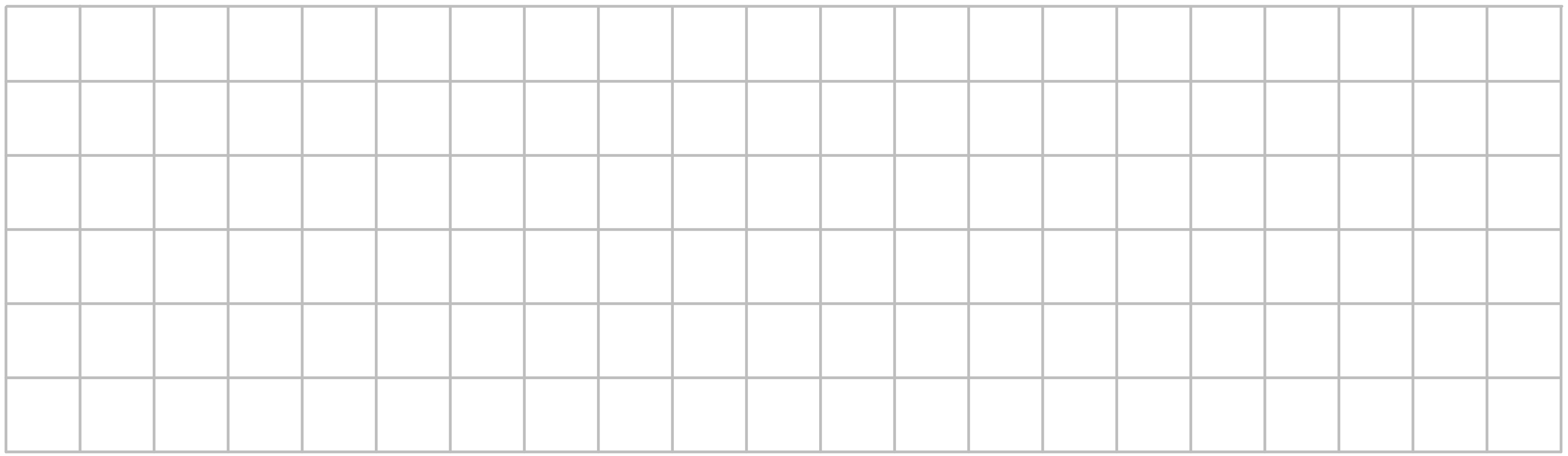
Y-cooridinates: depth of vertices
X-cooridinates: based on in-order tree traversal

Level-based layout A simple approach

Input: A binary tree T
Output: A leveled drawing of T

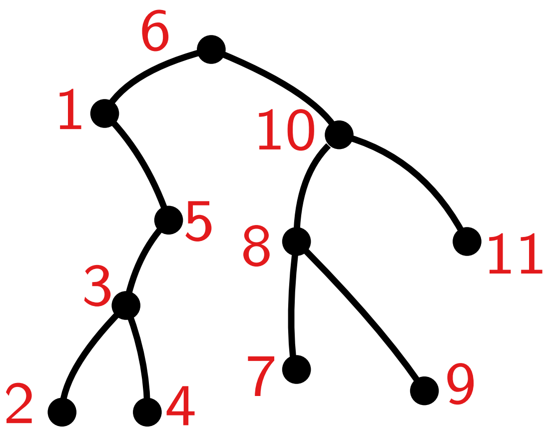


Y-cooridinates: depth of vertices
X-cooridinates: based on in-order tree traversal

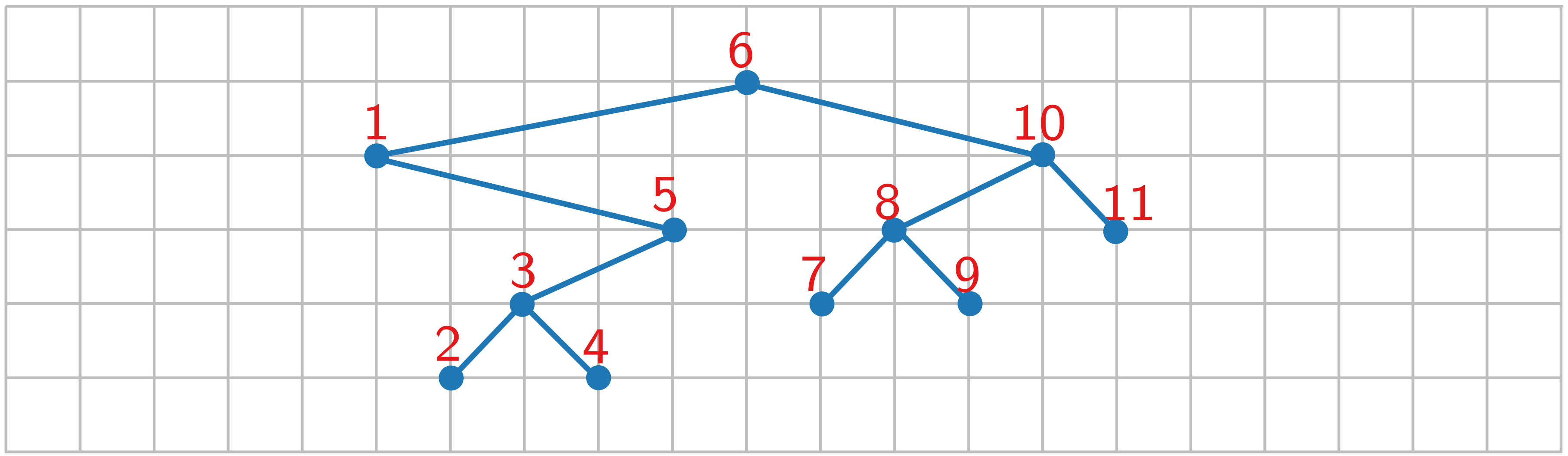


Level-based layout A simple approach

Input: A binary tree T
Output: A leveled drawing of T

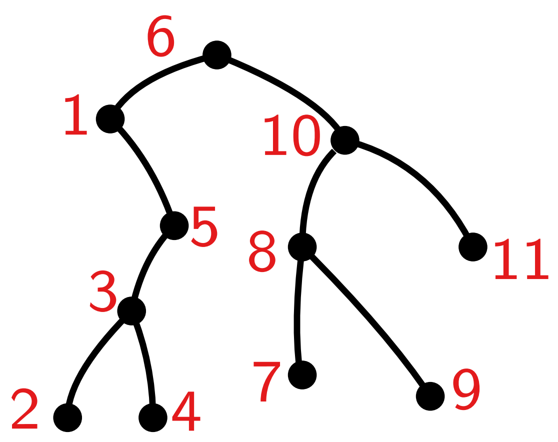


Y-cooridinates: depth of vertices
X-cooridinates: based on in-order tree traversal

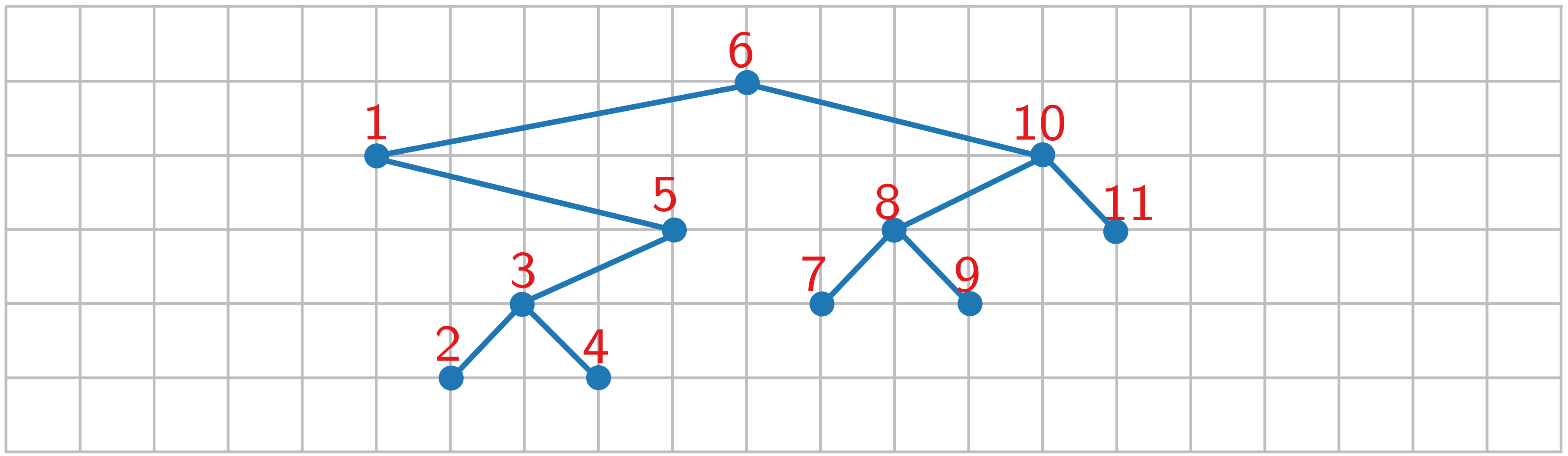


Level-based layout A simple approach

Input: A binary tree T
Output: A leveled drawing of T



Y-cooridinates: depth of vertices
X-cooridinates: based on in-order tree traversal

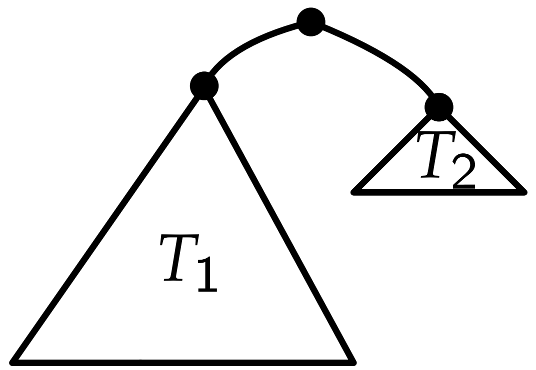


Issues:

- Drawing is wider than needed
- Parents not in the center of span of their children

Level-based layout: A divide and conquer approach

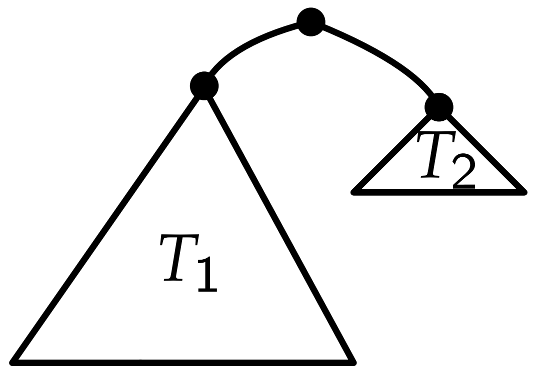
Input: A binary tree T
Output: A leveled drawing of T



Level-based layout: A divide and conquer approach

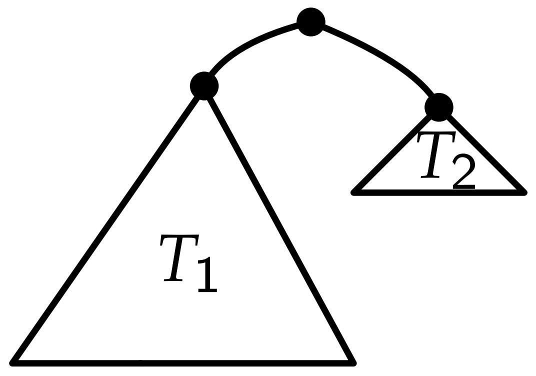
Input: A binary tree T
Output: A leveled drawing of T

Base case: A single vertex ●



Level-based layout: A divide and conquer approach

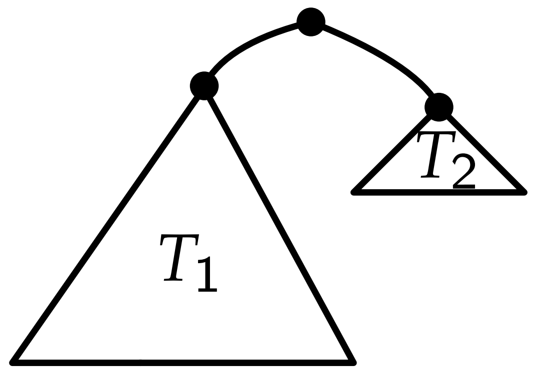
Input: A binary tree T
Output: A leveled drawing of T



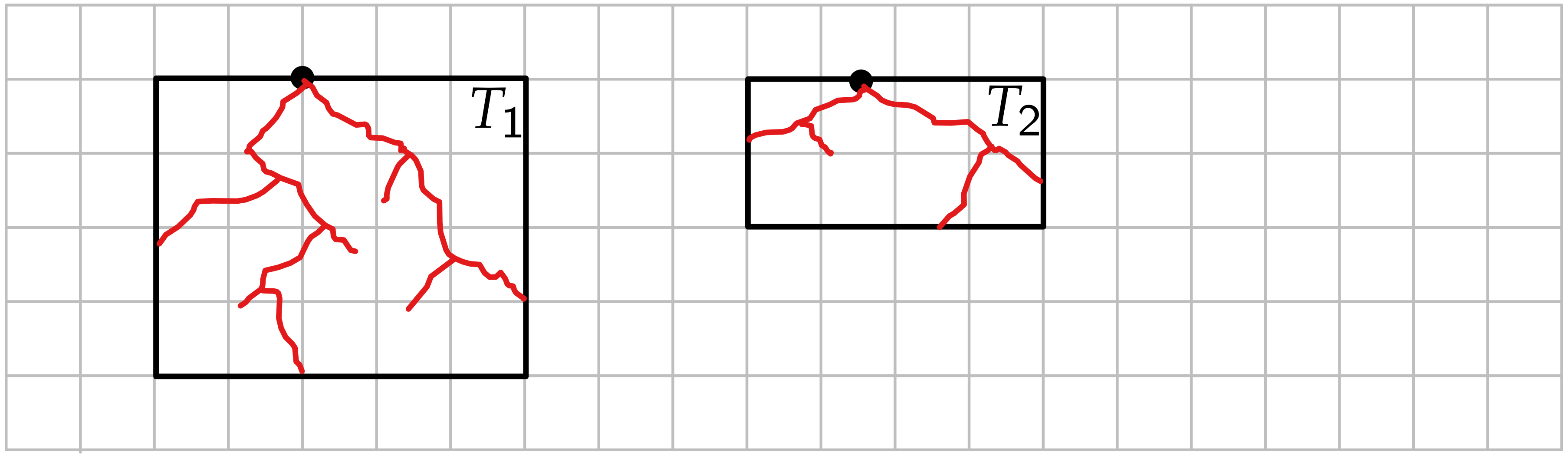
Base case: A single vertex ●
Divide: Recursively apply the algorithm to draw the left and right subtrees

Level-based layout: A divide and conquer approach

Input: A binary tree T
Output: A leveled drawing of T

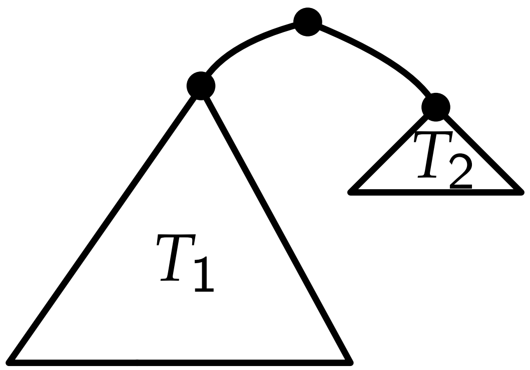


Base case: A single vertex ●
Divide: Recursively apply the algorithm to draw the left and right subtrees

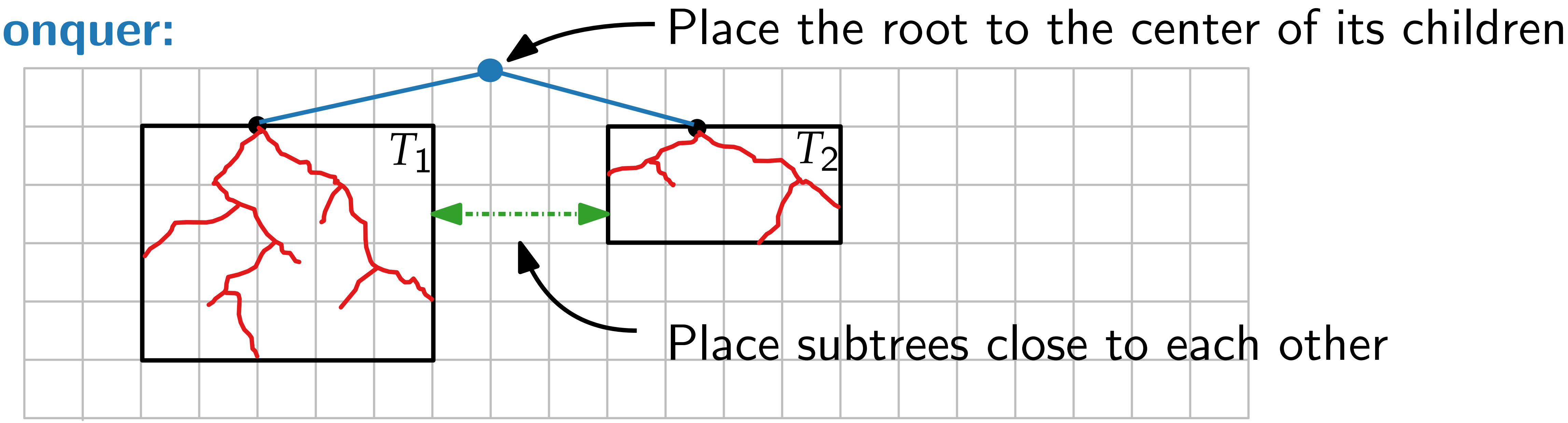


Level-based layout: A divide and conquer approach

Input: A binary tree T
Output: A leveled drawing of T

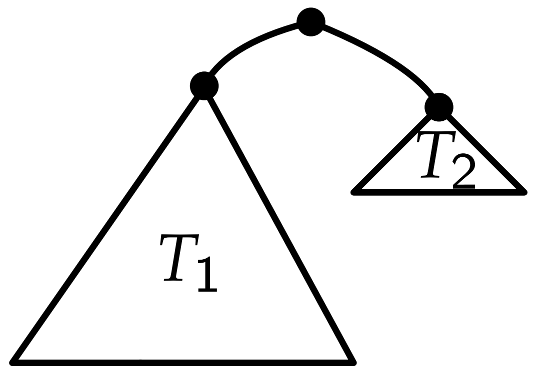


Base case: A single vertex ●
Divide: Recursively apply the algorithm to draw the left and right subtrees
Conquer:



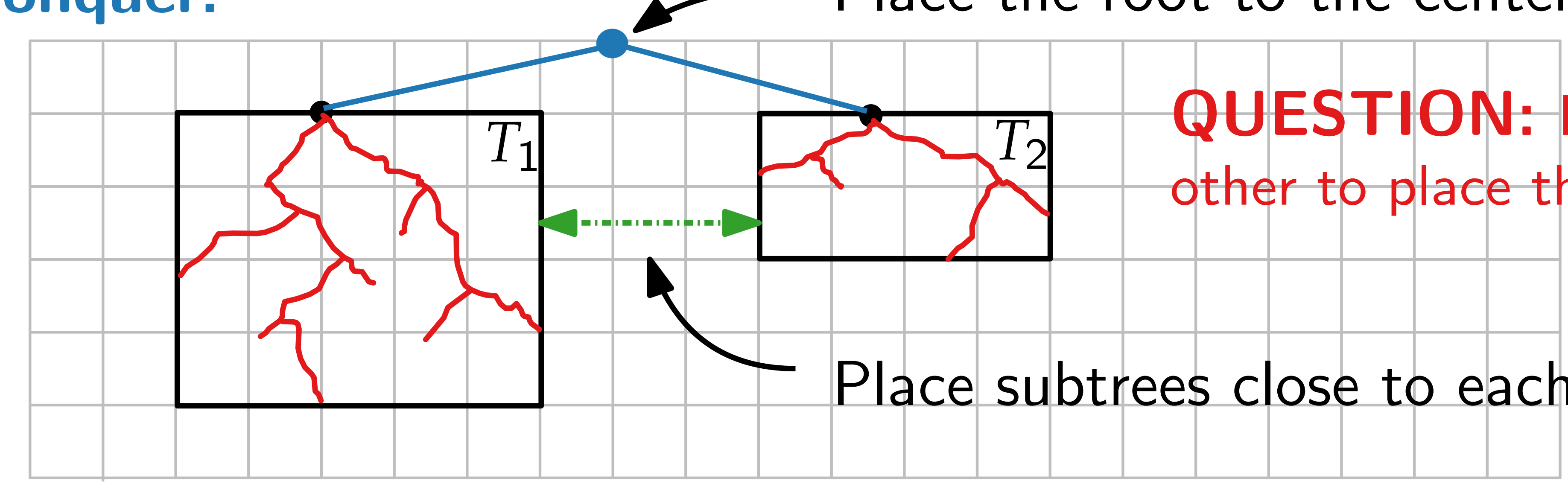
Level-based layout: A divide and conquer approach

Input: A binary tree T
Output: A leveled drawing of T



Base case: A single vertex ●
Divide: Recursively apply the algorithm to draw the left and right subtrees
Conquer:

Place the root to the center of its children

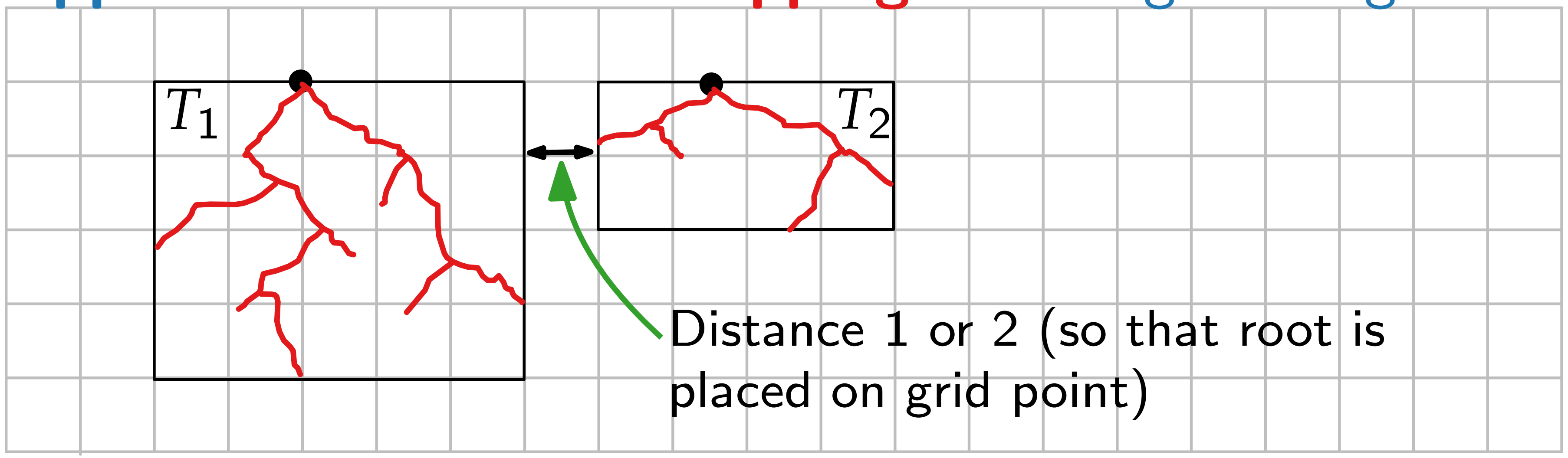


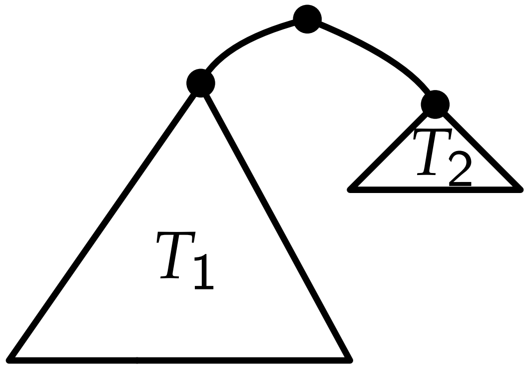
QUESTION: How close to each other to place the subtrees?

Place subtrees close to each other

Level-based layout: A divide and conquer approach

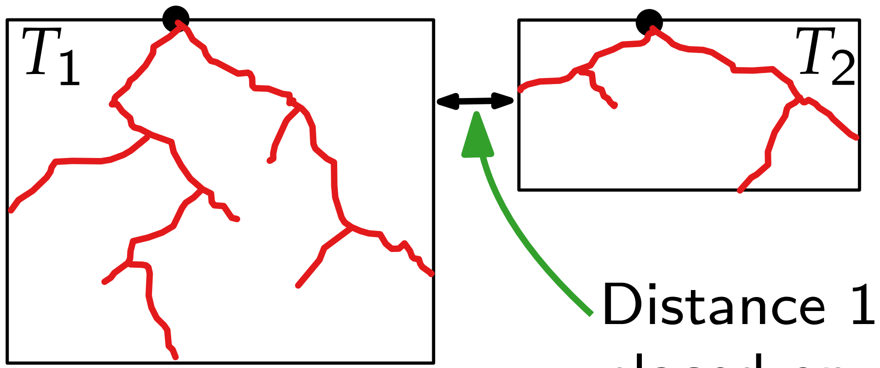
Approach-1: **Non-overlapping** enclosing rectangles



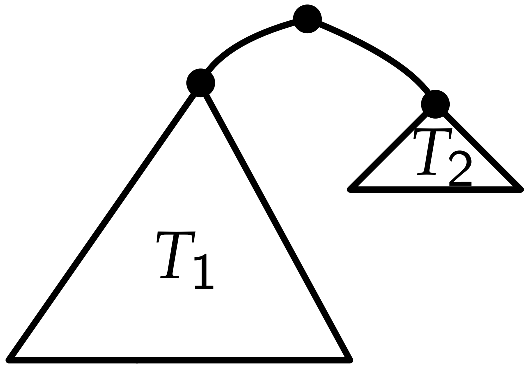


Level-based layout: A divide and conquer approach

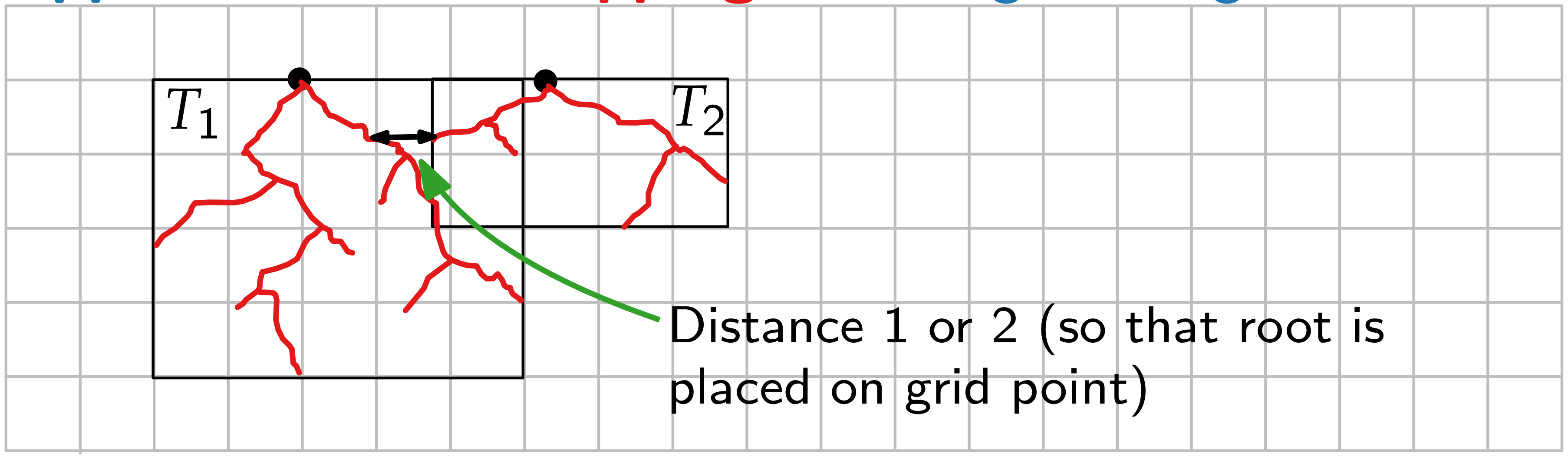
Approach-1: **Non-overlapping** enclosing rectangles



Distance 1 or 2 (so that root is placed on grid point)



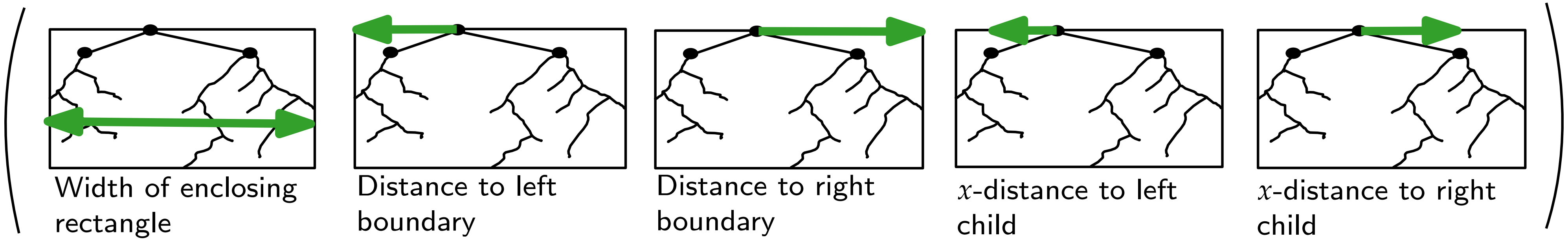
Approach-2: **Overlapping** enclosing rectangles



Distance 1 or 2 (so that root is placed on grid point)

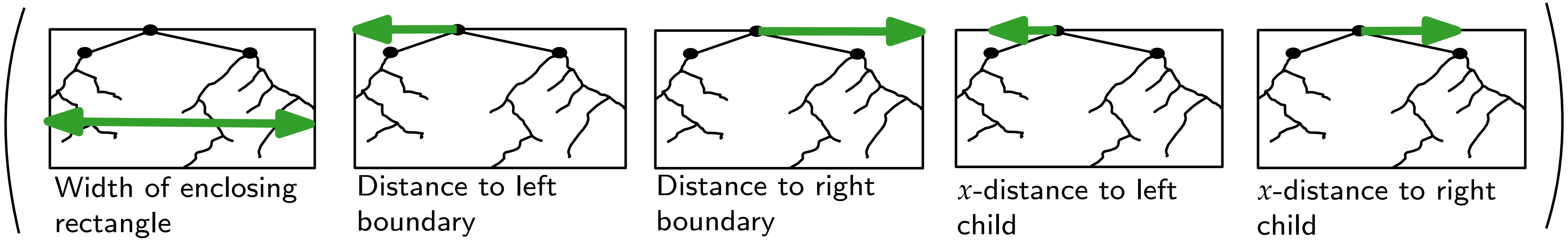
Implementation: Non-overlapping rectangles

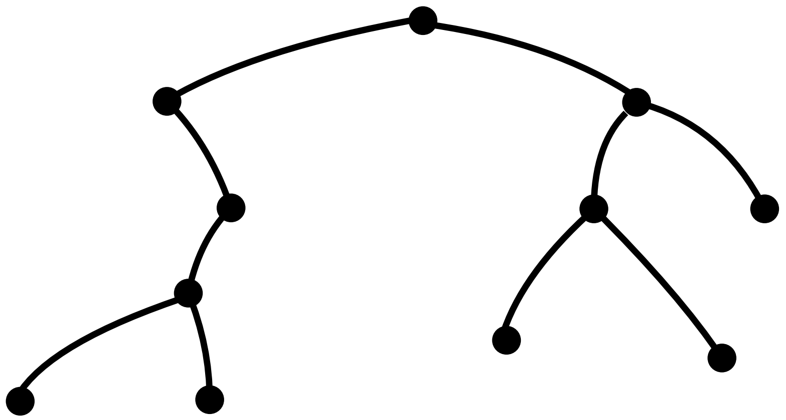
- In a bottom up manner (by a post-order traversal) we compute for each vertex the 5-tuple:



Implementation: Non-overlapping rectangles

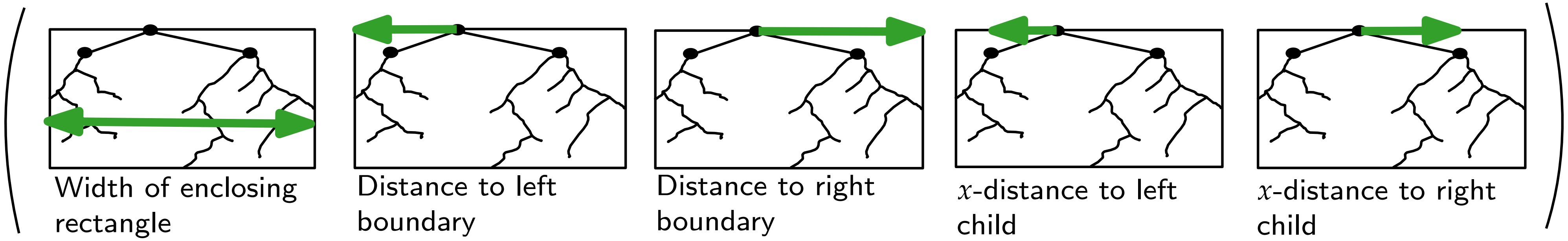
- In a bottom up manner (by a post-order traversal) we compute for each vertex the 5-tuple:



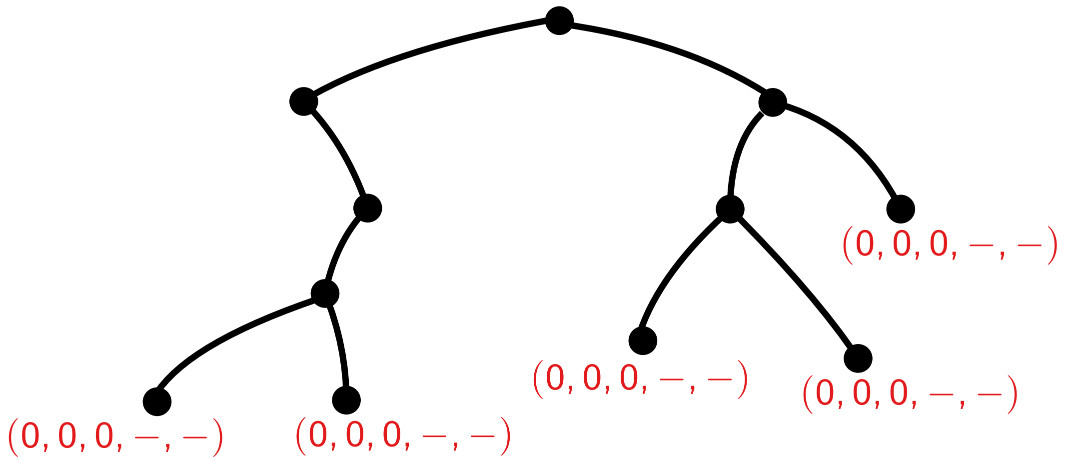


Implementation: Non-overlapping rectangles

- In a bottom up manner (by a post-order traversal) we compute for each vertex the 5-tuple:

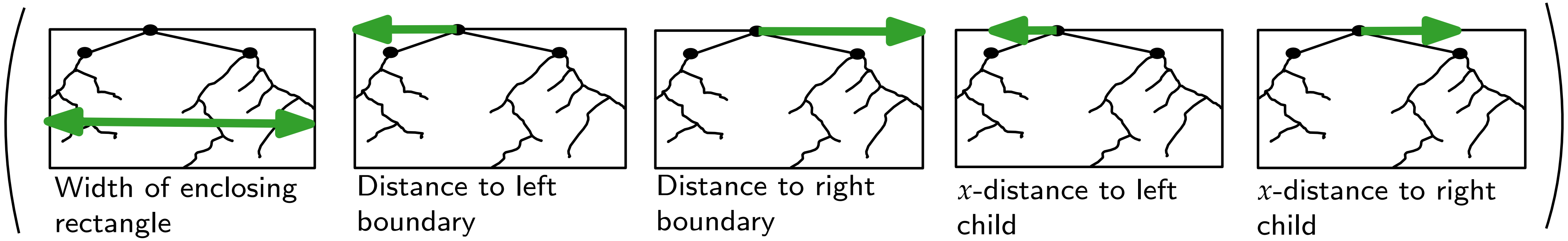


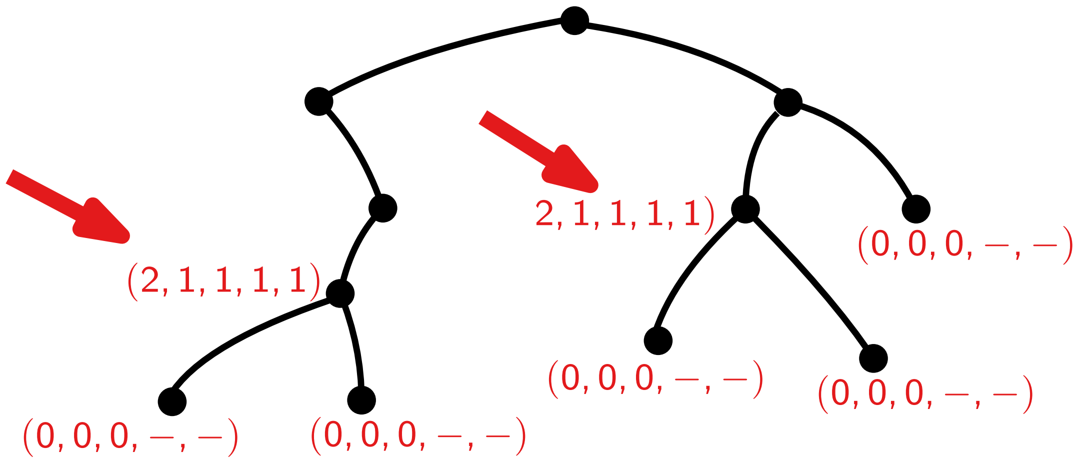
- For leaves: $(0, 0, 0, -, -)$



Implementation: Non-overlapping rectangles

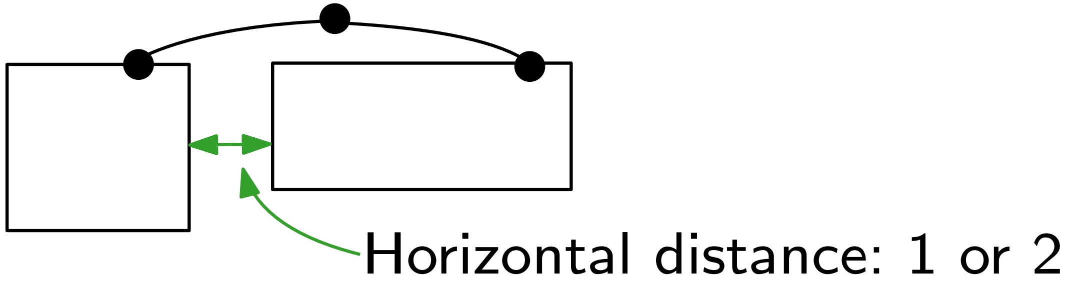
- In a bottom up manner (by a post-order traversal) we compute for each vertex the 5-tuple:





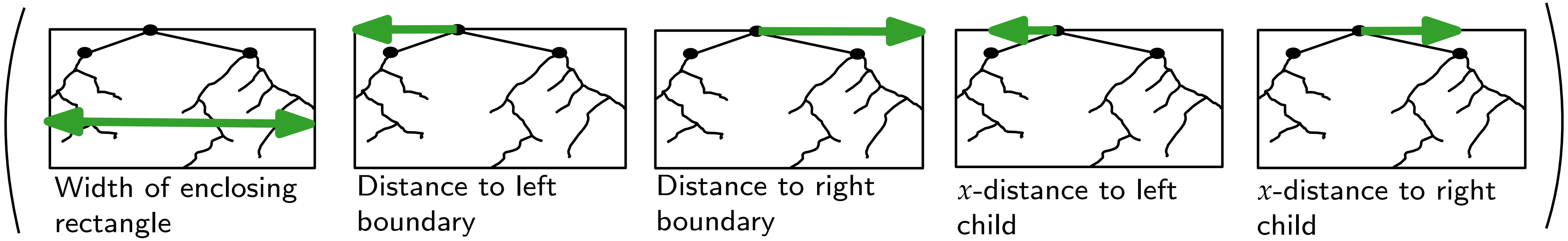
Rule-1:

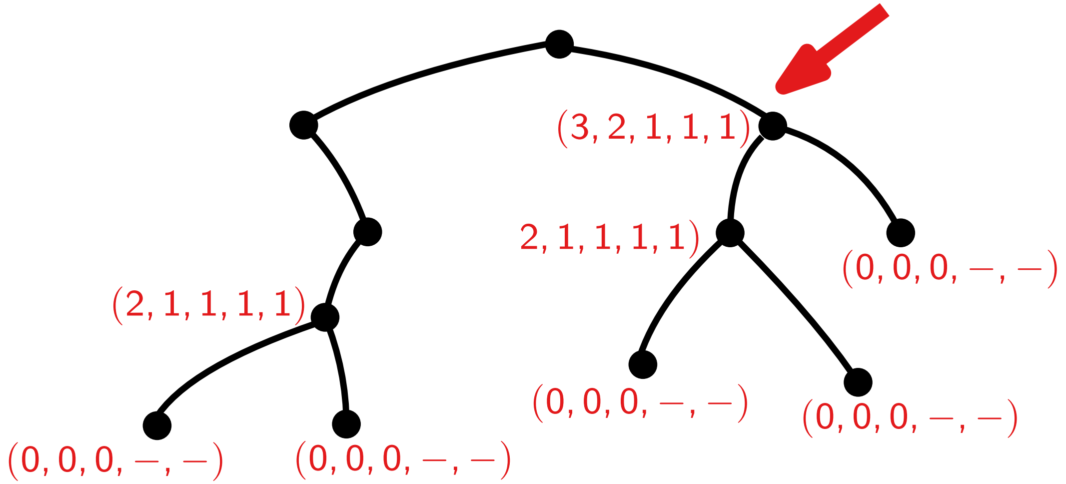
- Parent centered above children
- Parent at grid point



Implementation: Non-overlapping rectangles

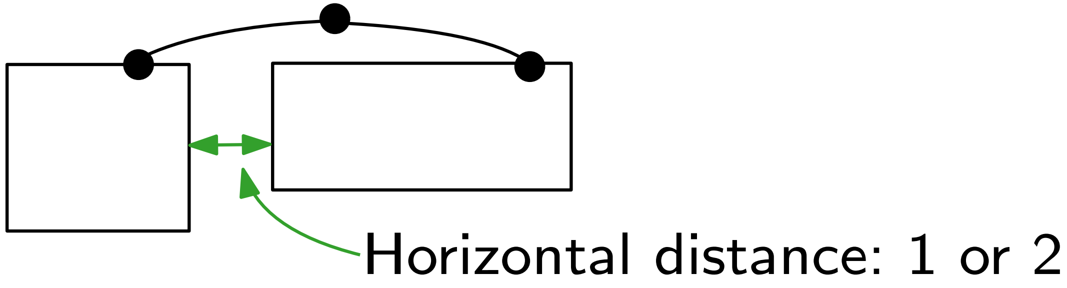
- In a bottom up manner (by a post-order traversal) we compute for each vertex the 5-tuple:





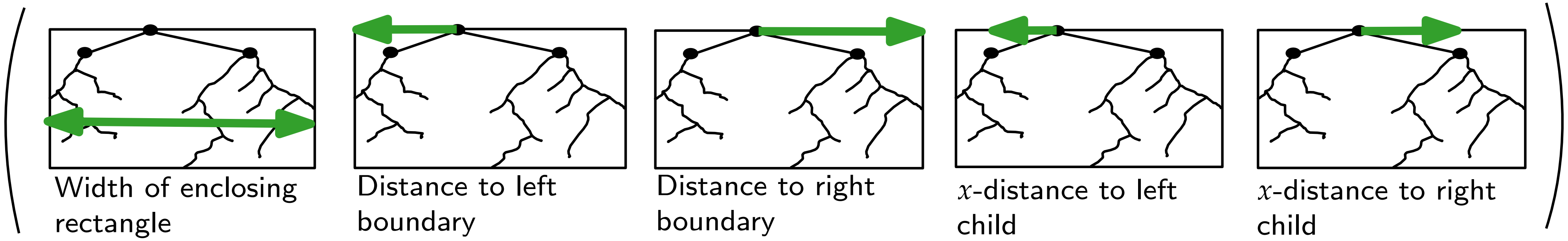
Rule-1:

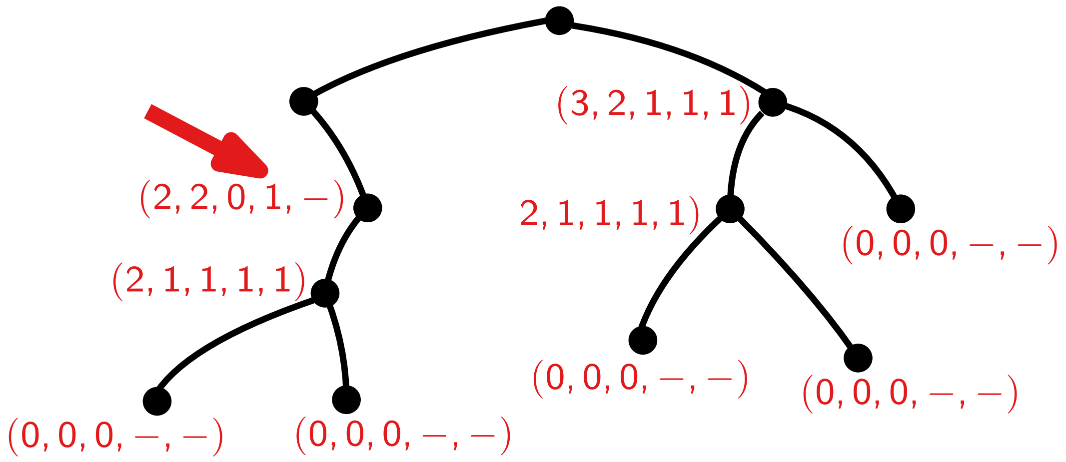
- Parent centered above children
- Parent at grid point



Implementation: Non-overlapping rectangles

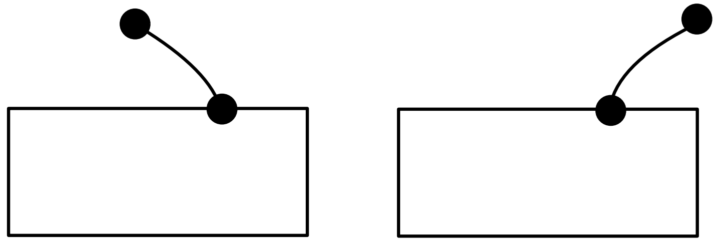
- In a bottom up manner (by a post-order traversal) we compute for each vertex the 5-tuple:





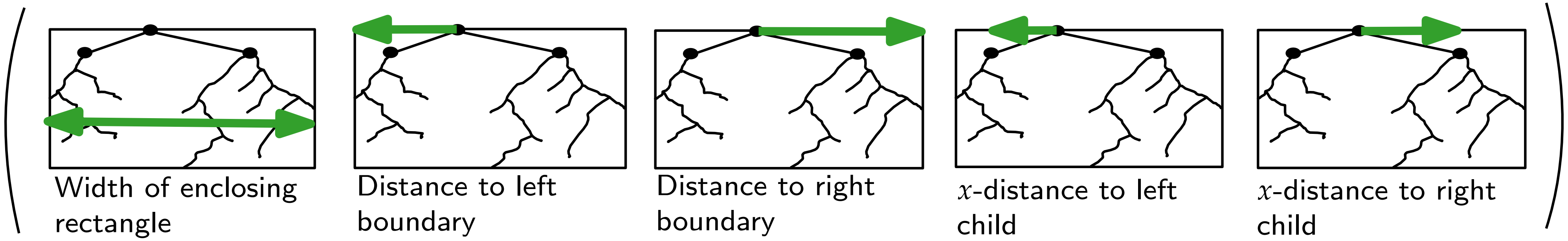
Rule-2:

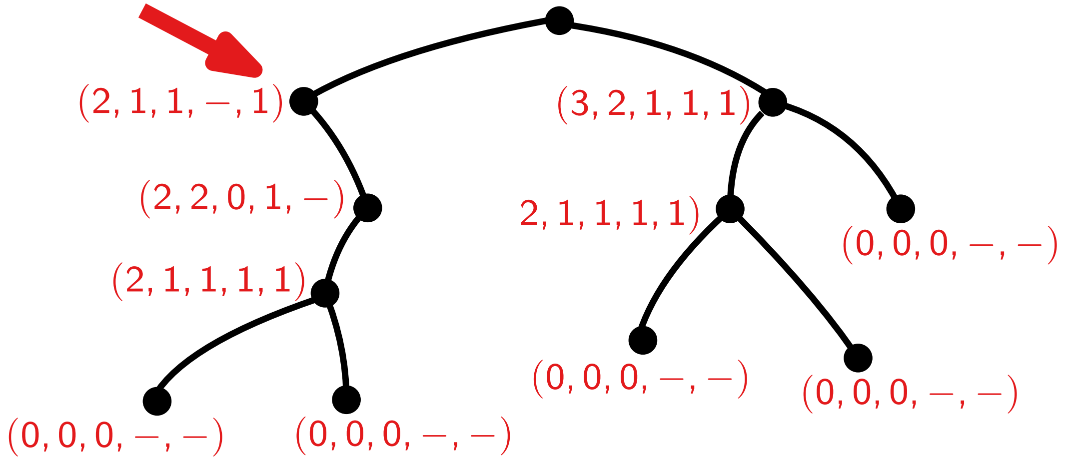
- Parent above and one unit to the left/right of single child



Implementation: Non-overlapping rectangles

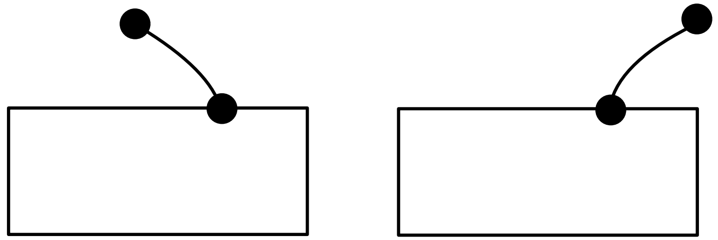
- In a bottom up manner (by a post-order traversal) we compute for each vertex the 5-tuple:





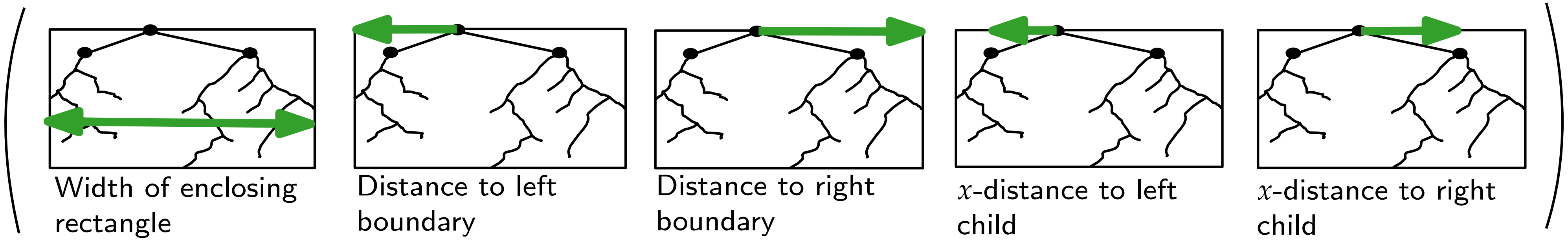
Rule-2:

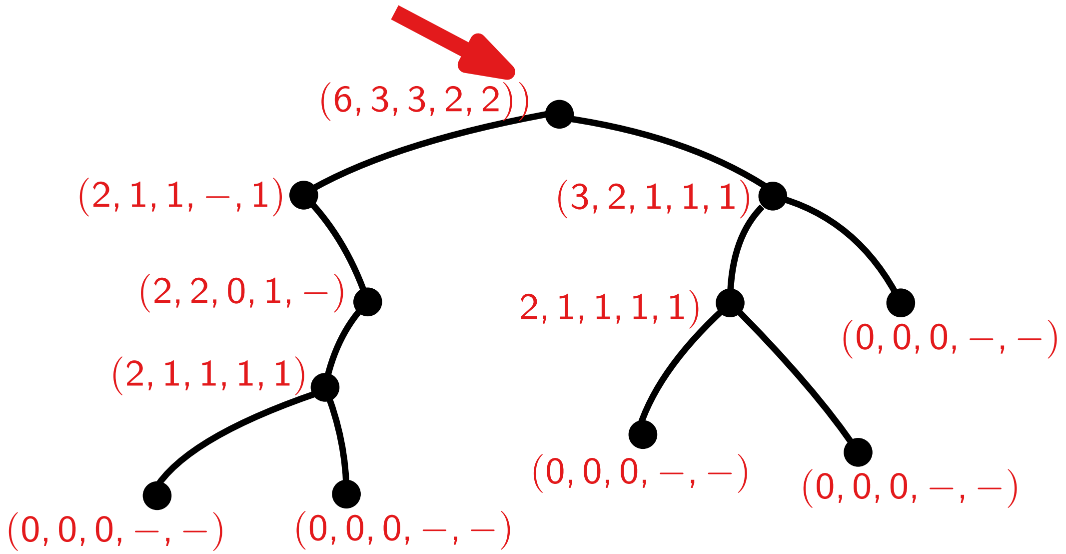
- Parent above and one unit to the left/right of single child



Implementation: Non-overlapping rectangles

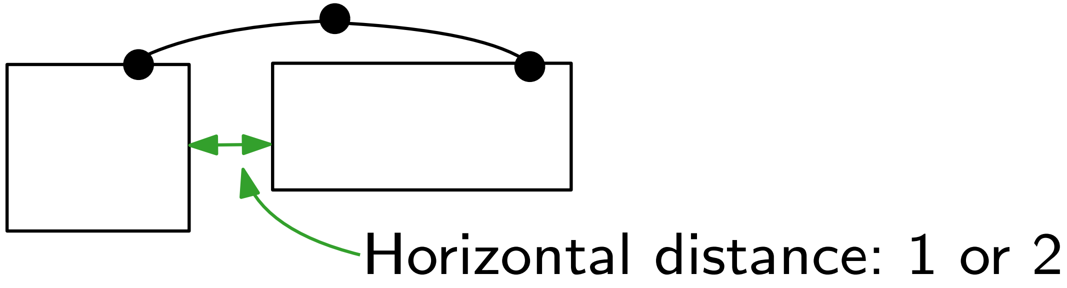
- In a bottom up manner (by a post-order traversal) we compute for each vertex the 5-tuple:





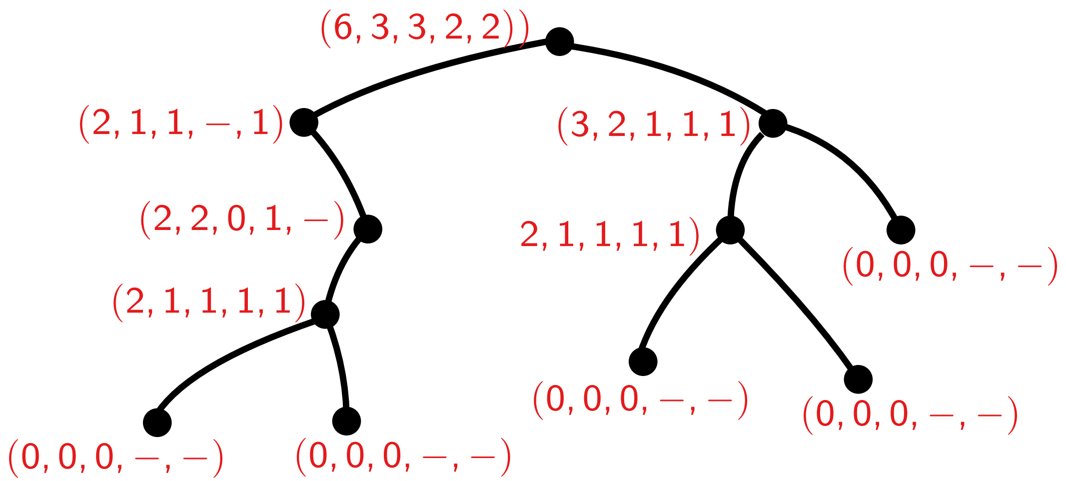
Rule-1:

- Parent centered above children
- Parent at grid point



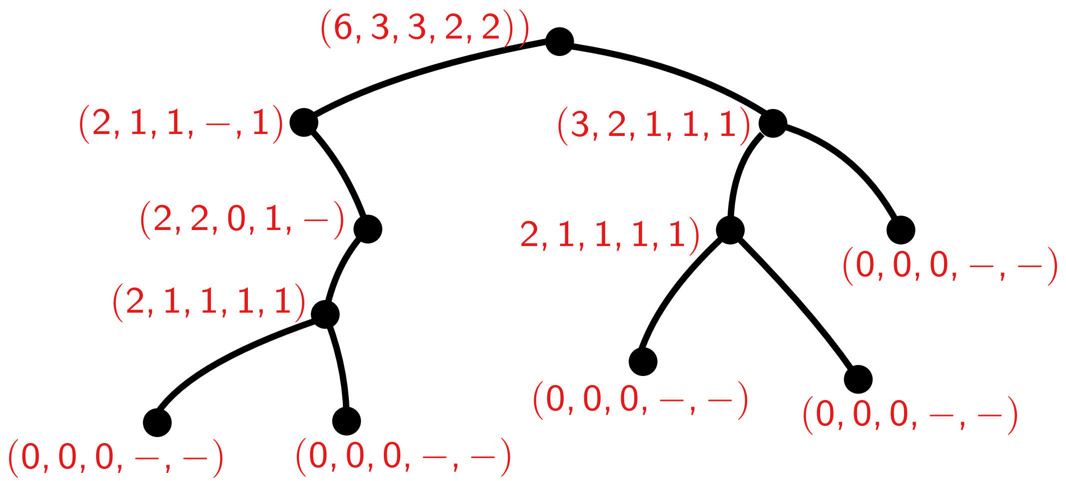
Implementation: Non-overlapping rectangles

- **Computation of x-coordinates by pre-order traversal**



Implementation: Non-overlapping rectangles

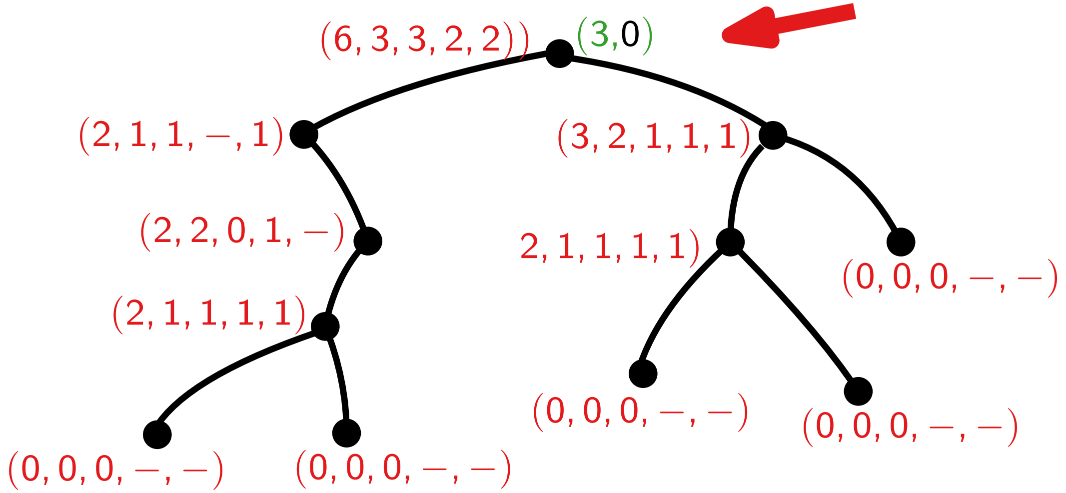
- **Computation of x-coordinates by pre-order traversal**



- **y-coordinate:** the depth of each node

Implementation: Non-overlapping rectangles

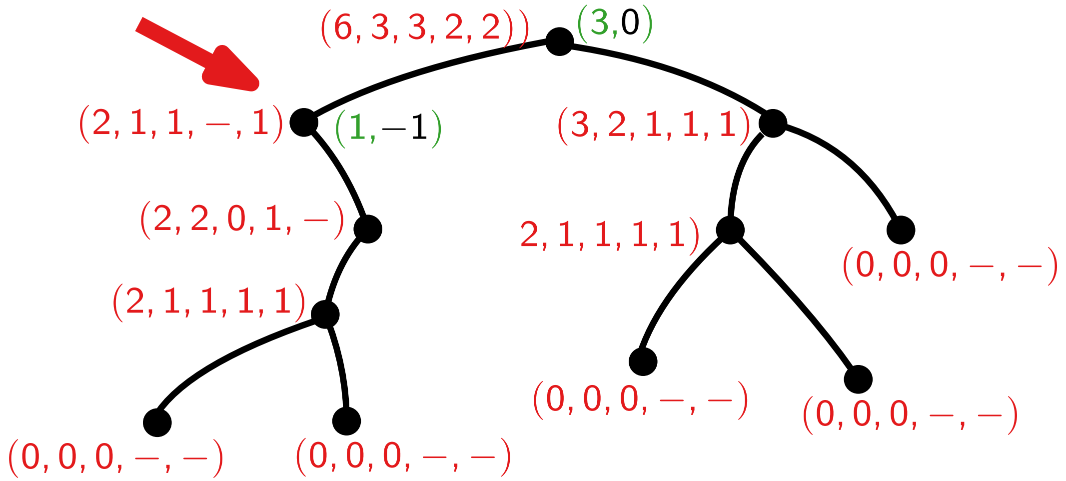
- **Computation of x-coordinates by pre-order traversal**



- **y-coordinate:** the depth of each node

Implementation: Non-overlapping rectangles

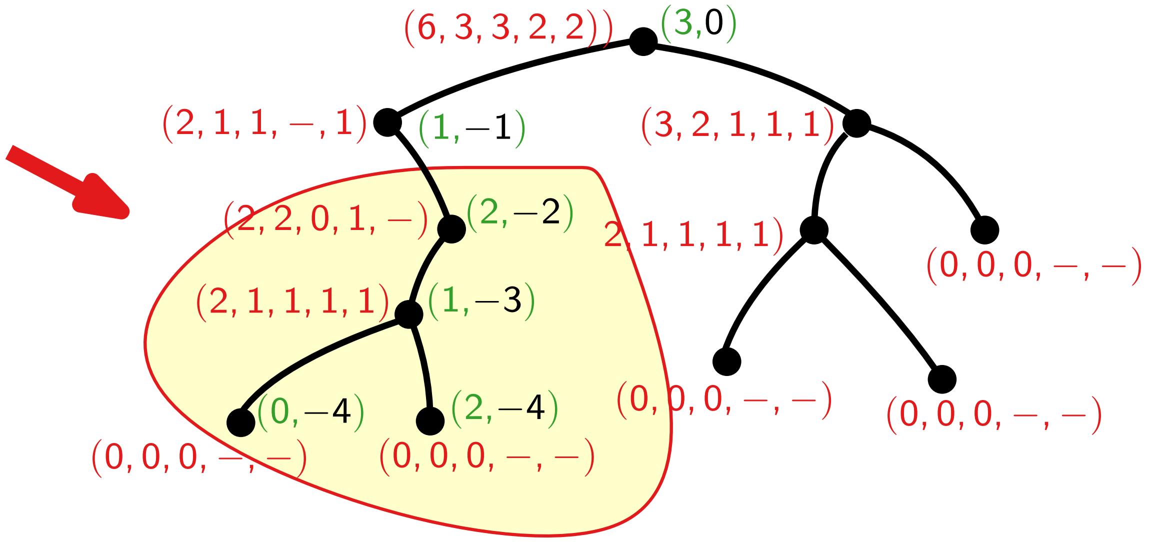
- **Computation of x-coordinates by pre-order traversal**



- **y-coordinate:** the depth of each node

Implementation: Non-overlapping rectangles

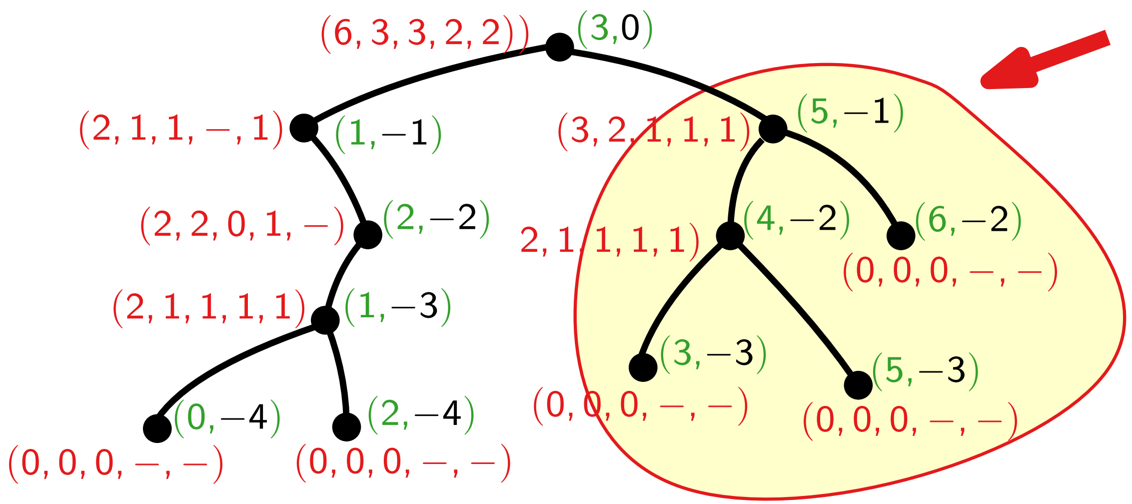
- **Computation of x-coordinates by pre-order traversal**



- **y-coordinate:** the depth of each node

Implementation: Non-overlapping rectangles

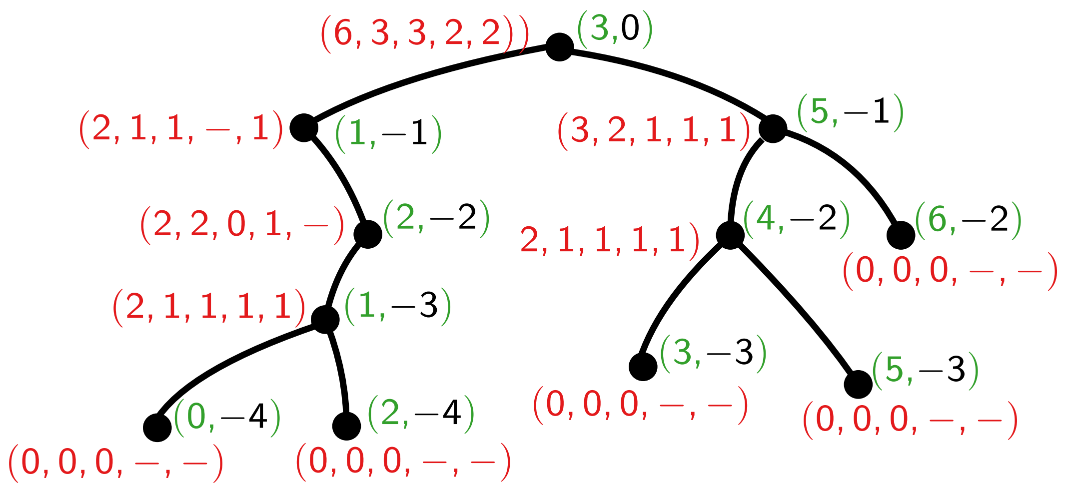
- **Computation of x-coordinates by pre-order traversal**



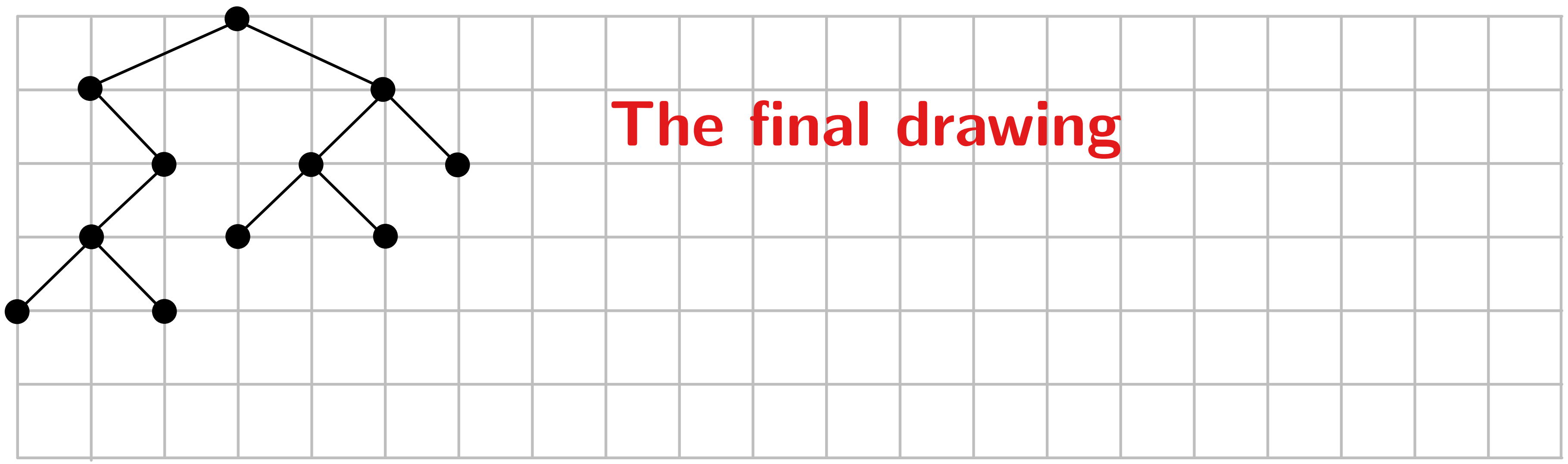
- **y-coordinate:** the depth of each node

Implementation: Non-overlapping rectangles

- **Computation of x-coordinates by pre-order traversal**



- **y-coordinate:** the depth of each node

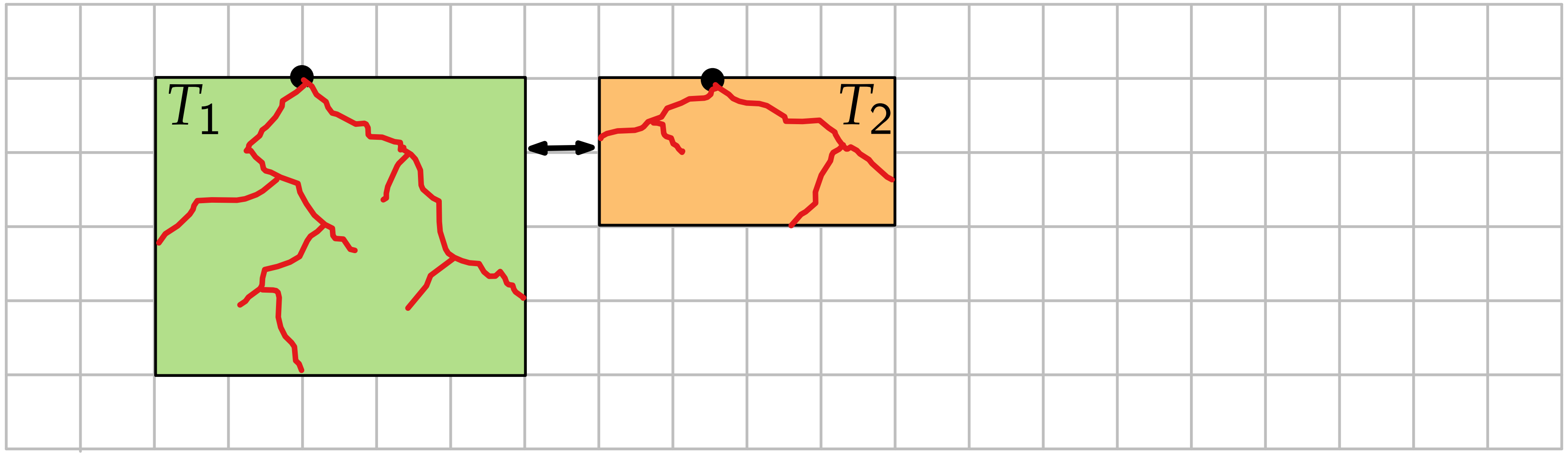


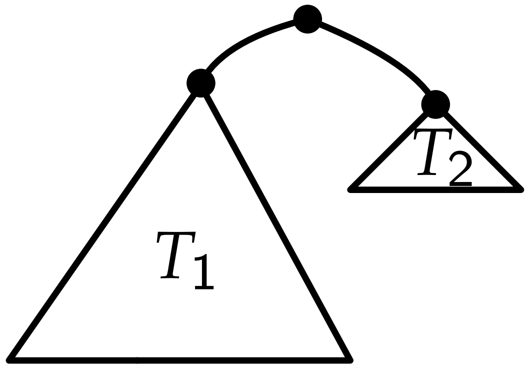
The final drawing

Implementation: Overlapping rectangles

Recall...

Approach-1: **Non-overlapping** enclosing rectangles



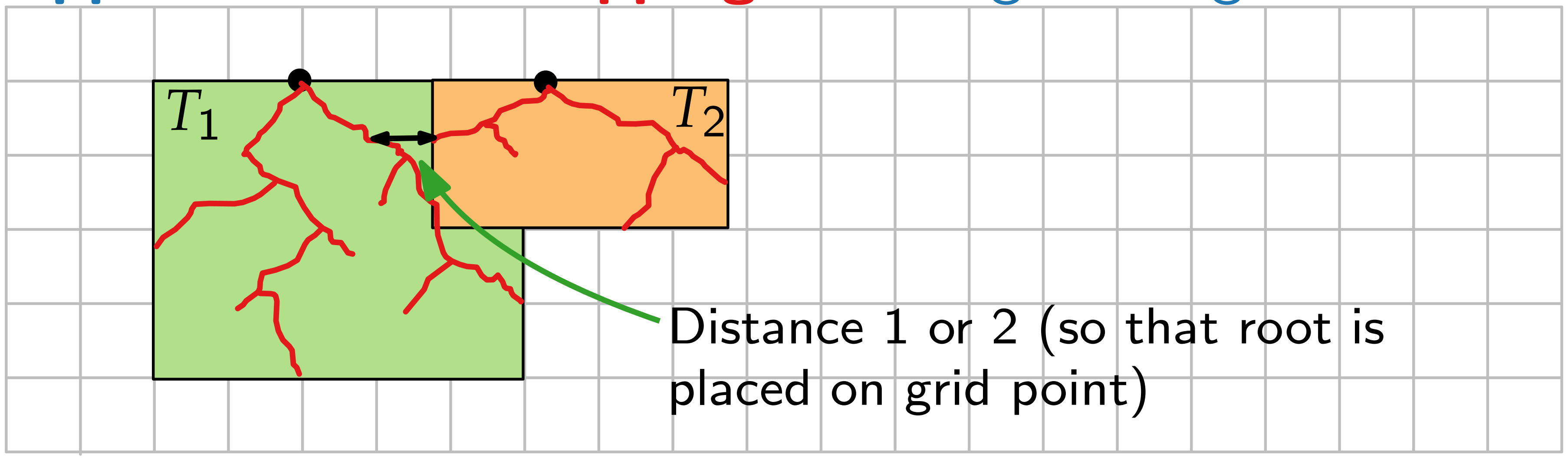


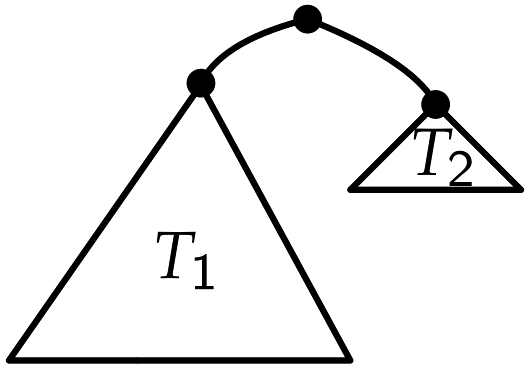
Implementation: Overlapping rectangles

Recall...

Approach-1: **Non-overlapping** enclosing rectangles

Approach-2: **Overlapping** enclosing rectangles

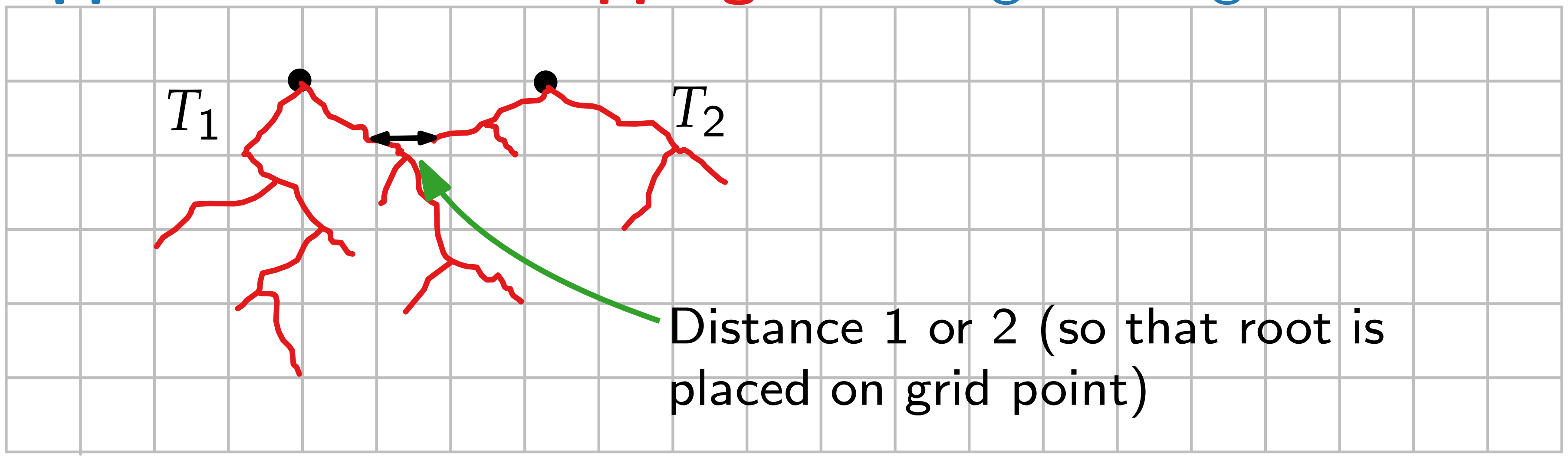


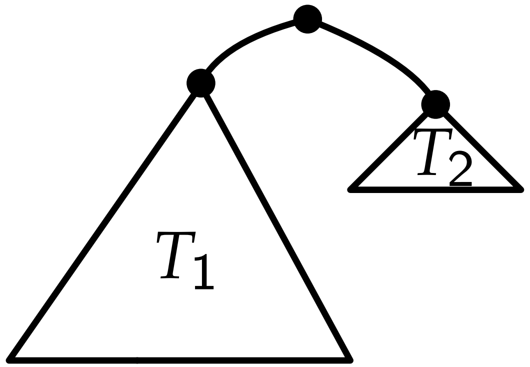


Implementation: Overlapping rectangles

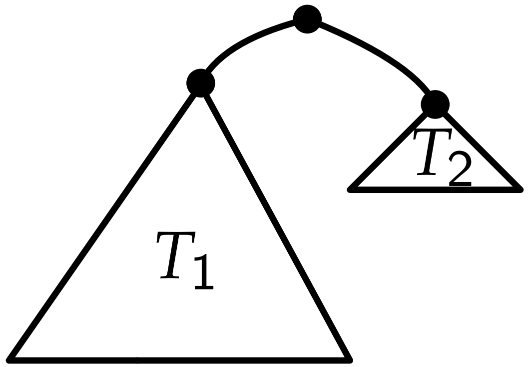
Recall...

Approach-1: **Non-overlapping** enclosing rectangles
Approach-2: **Overlapping** enclosing rectangles



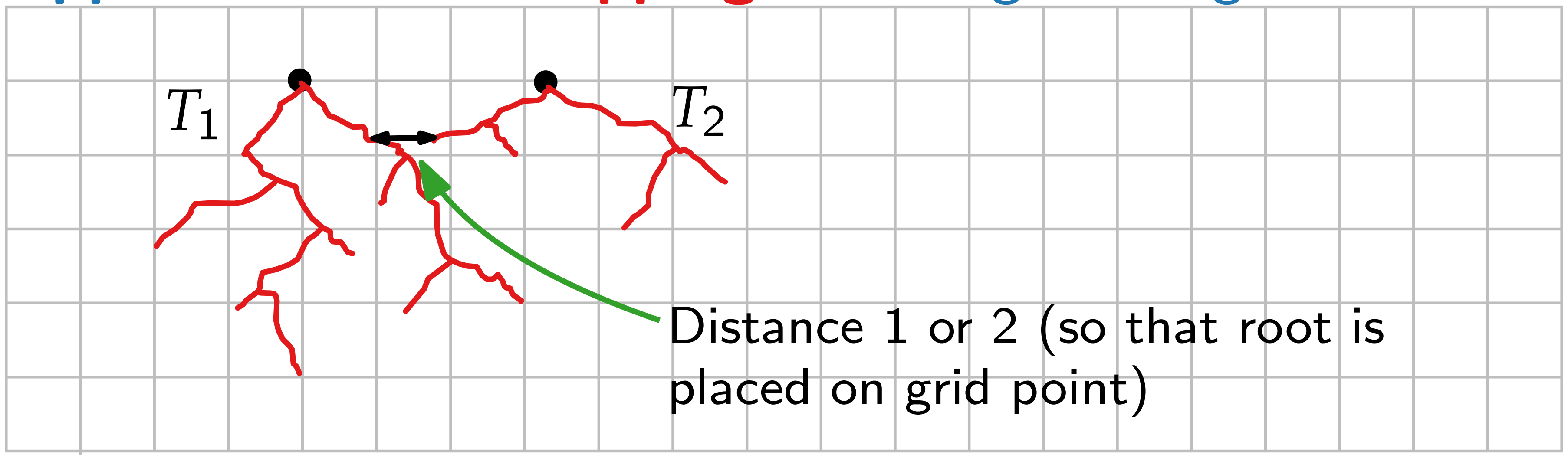


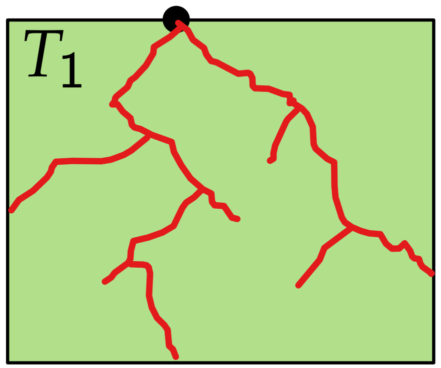
Implementation: Overlapping rectangles



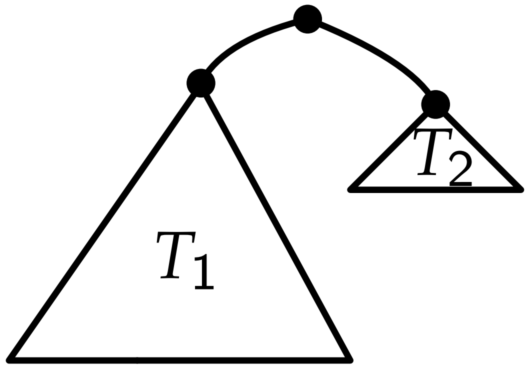
Recall...

Approach-1: **Non-overlapping** enclosing rectangles
Approach-2: **Overlapping** enclosing rectangles



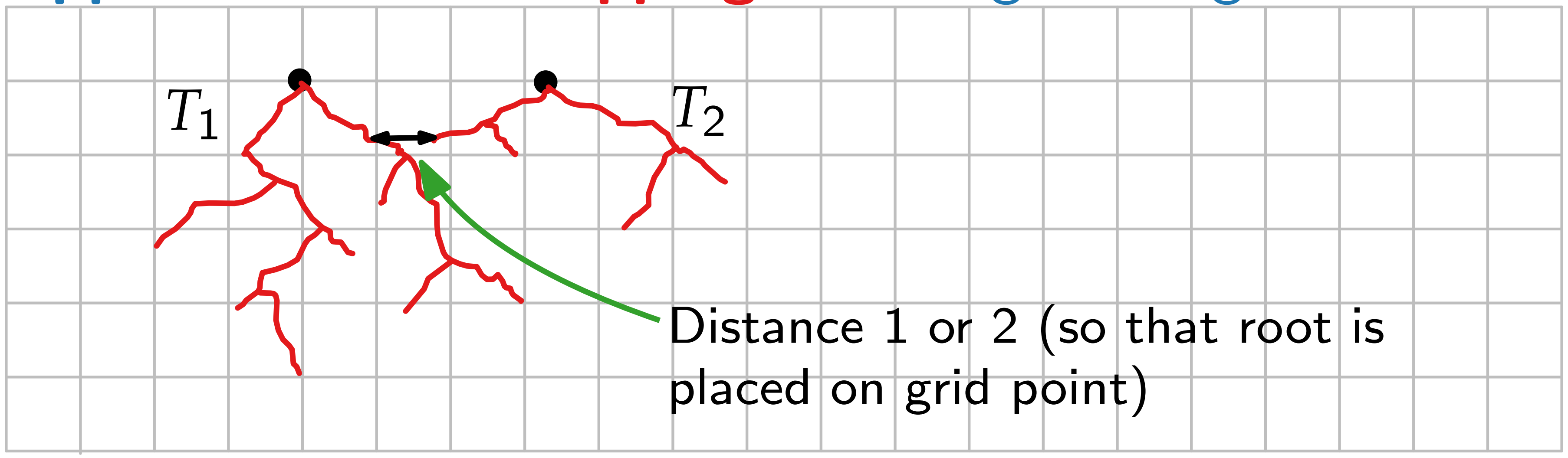


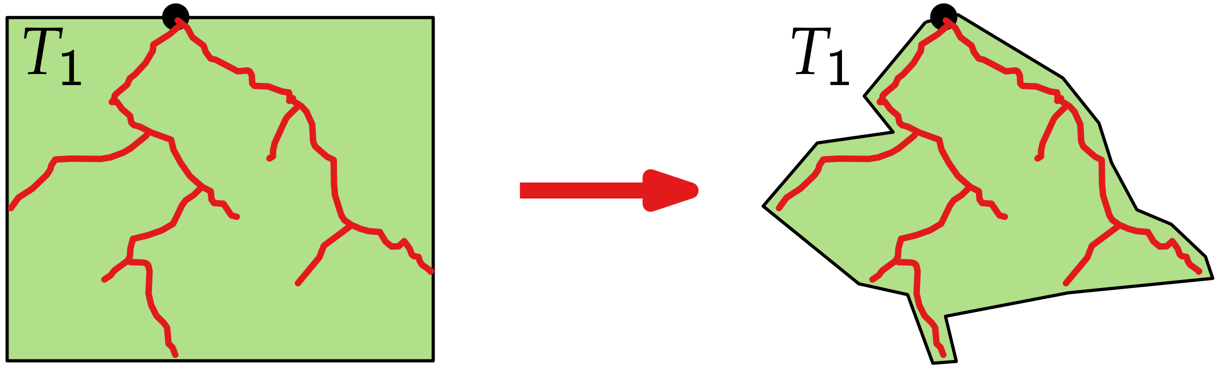
Implementation: Overlapping rectangles



Recall...

Approach-1: **Non-overlapping** enclosing rectangles
Approach-2: **Overlapping** enclosing rectangles

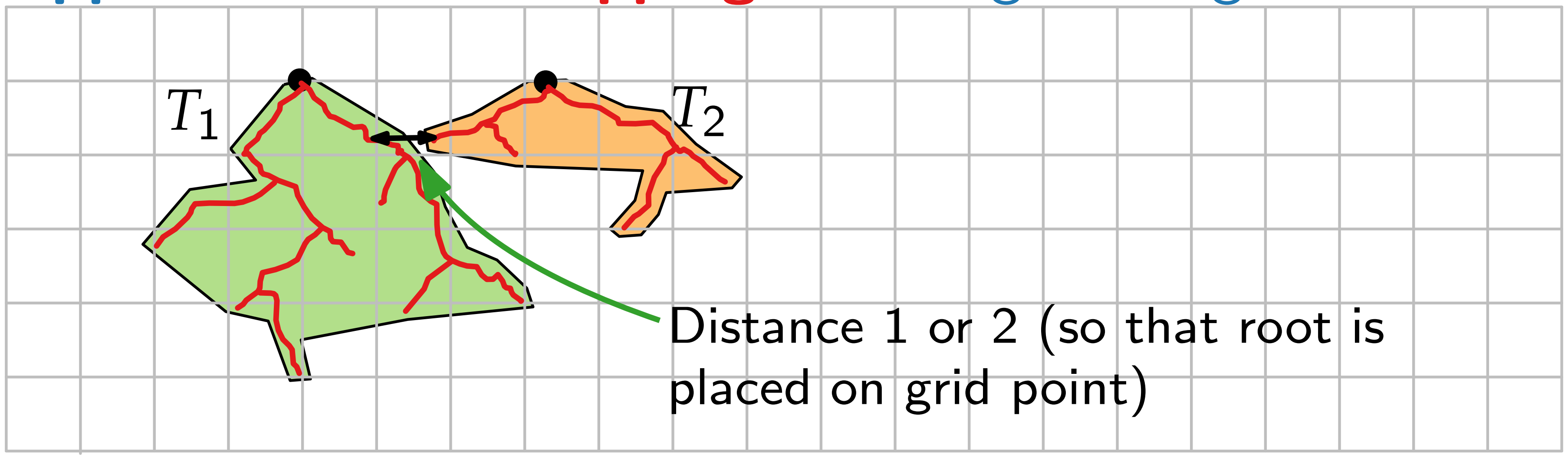


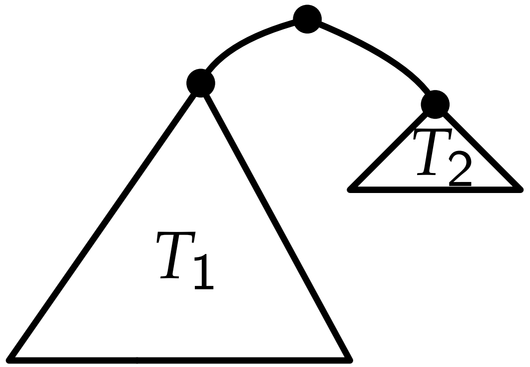


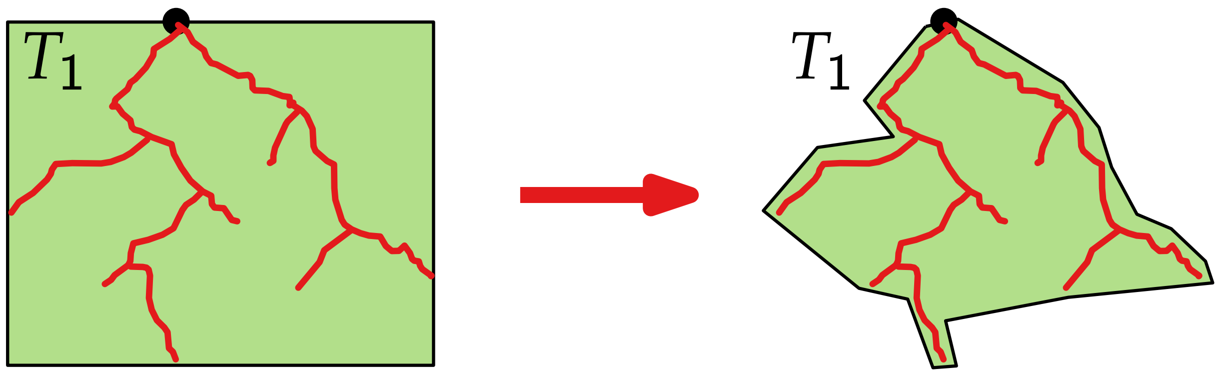
Implementation: Overlapping rectangles

Recall...

Approach-1: **Non-overlapping** enclosing rectangles
Approach-2: **Overlapping** enclosing rectangles



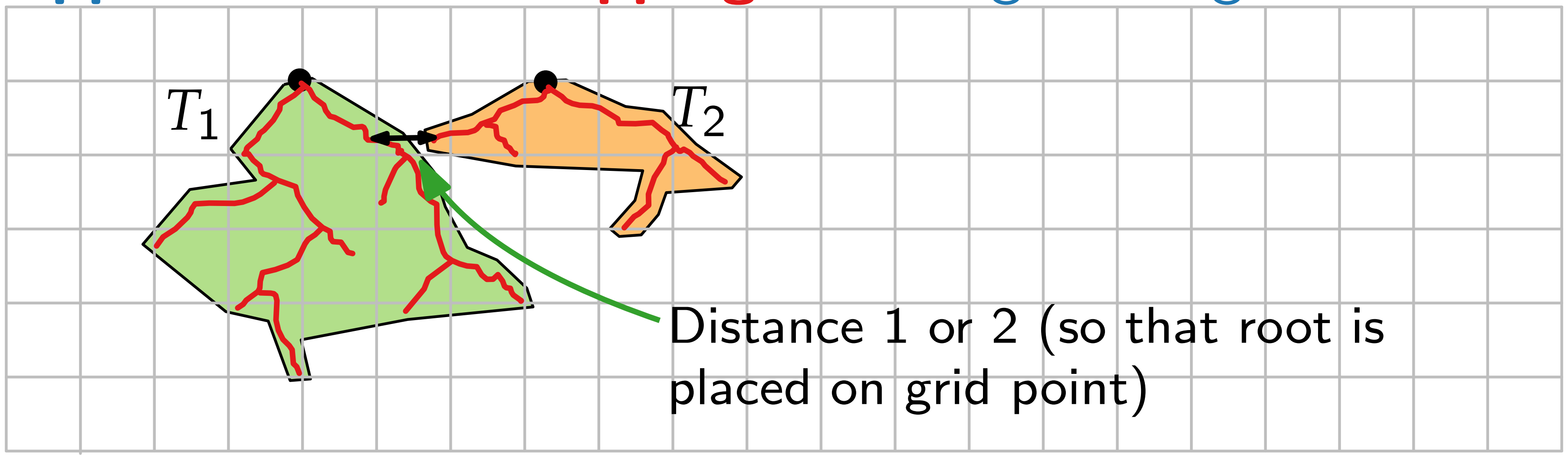


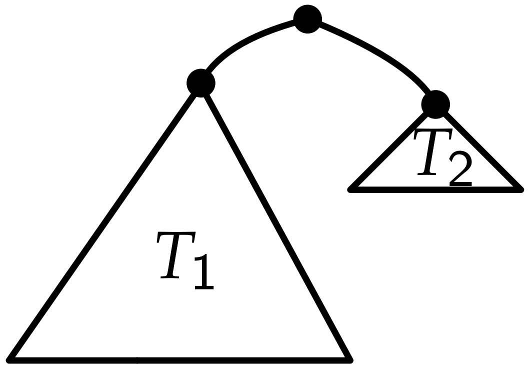


Implementation: Overlapping rectangles

Recall...

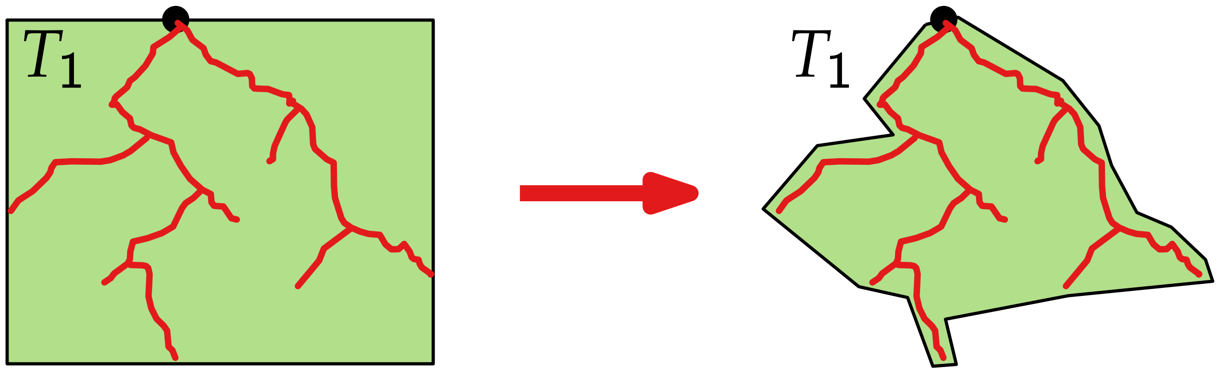
Approach-1: **Non-overlapping** enclosing rectangles
Approach-2: **Overlapping** enclosing rectangles





rectangles

contour



Implementation: Overlapping rectangles

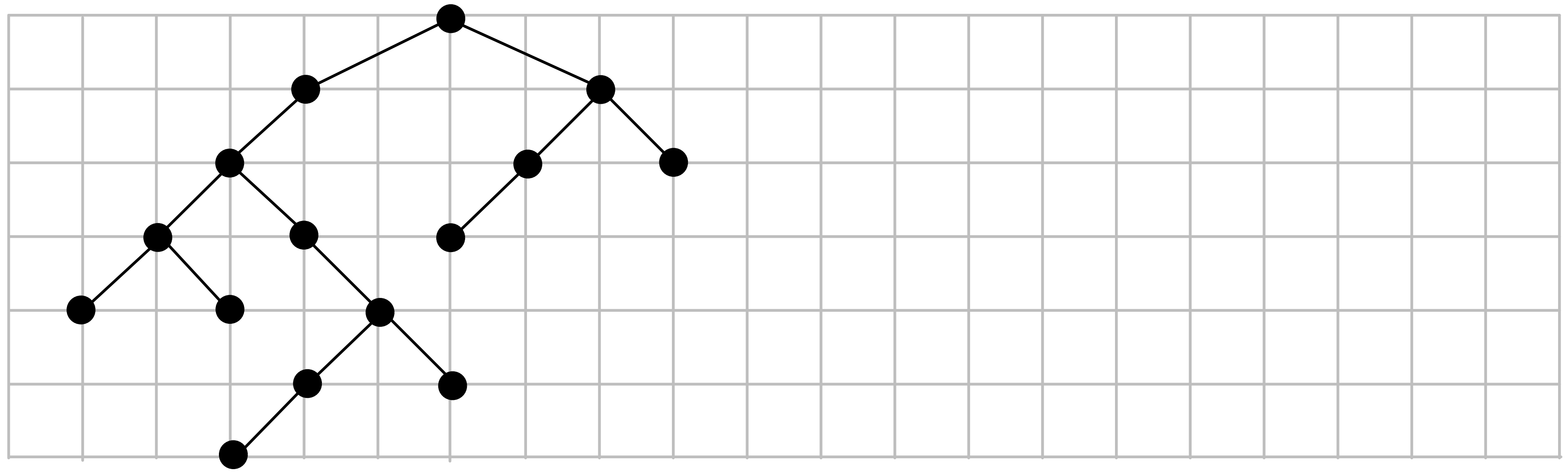
The left/right contour of leveled tree drawing

The *left/right contour* of a leveled tree drawing of height h is the sequence of vertices $(v_0, \ldots, v_h)$ such that vertex v_i is the leftmost/rightmost vertex at depth i

Implementation: Overlapping rectangles

The left/right contour of leveled tree drawing

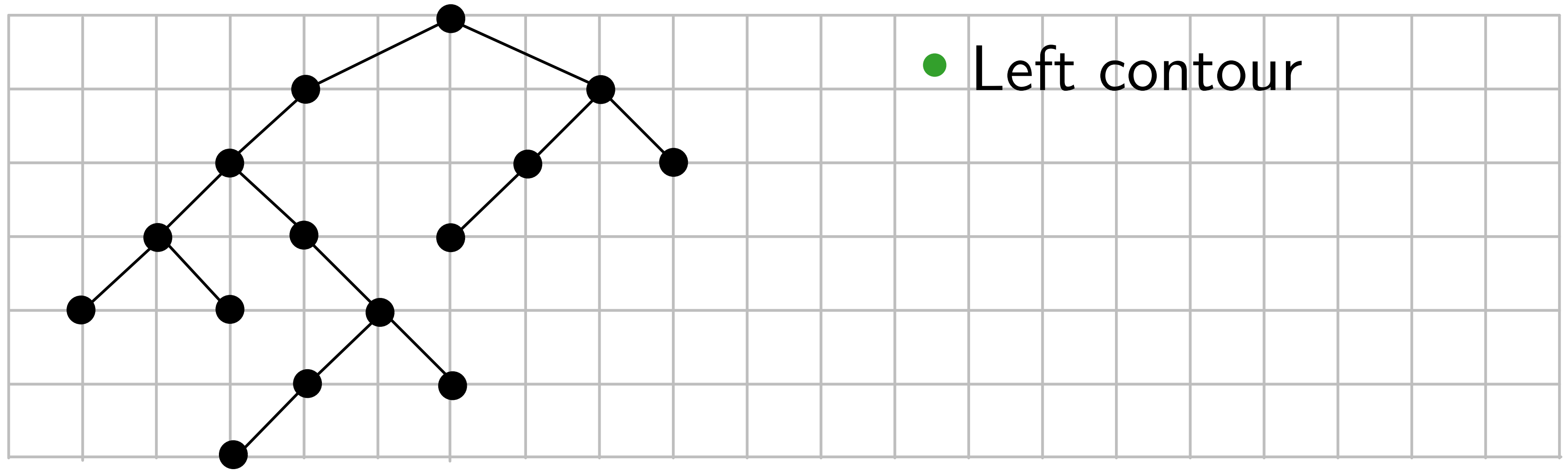
The *left/right contour* of a leveled tree drawing of height h is the sequence of vertices $(v_0, \ldots, v_h)$ such that vertex v_i is the leftmost/rightmost vertex at depth i



Implementation: Overlapping rectangles

The left/right contour of leveled tree drawing

The *left/right contour* of a leveled tree drawing of height h is the sequence of vertices $(v_0, \ldots, v_h)$ such that vertex v_i is the leftmost/rightmost vertex at depth i

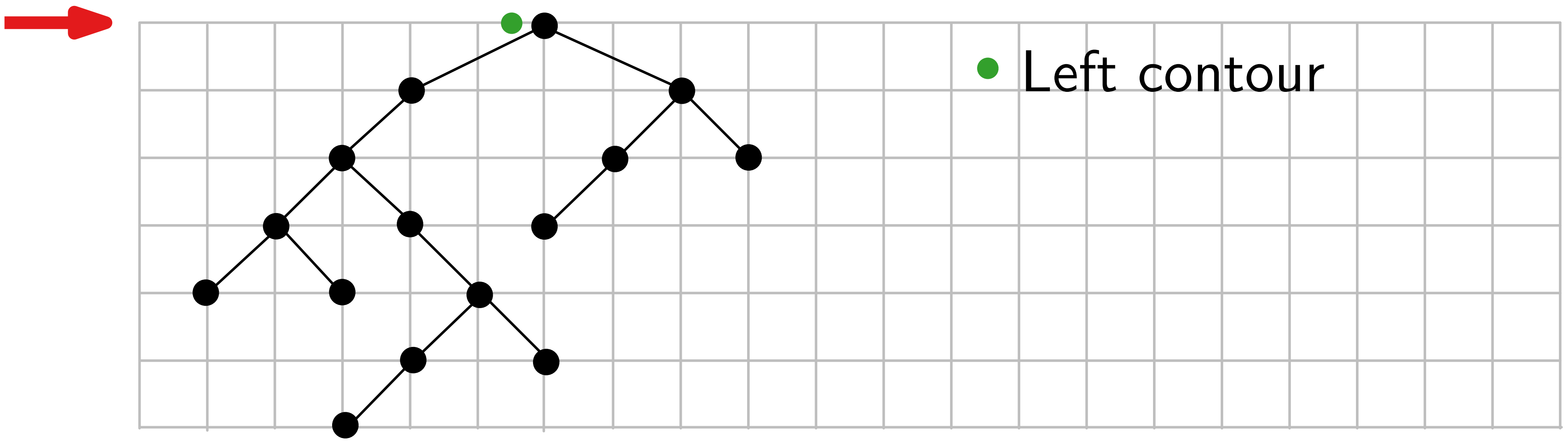


- Left contour

Implementation: Overlapping rectangles

The left/right contour of leveled tree drawing

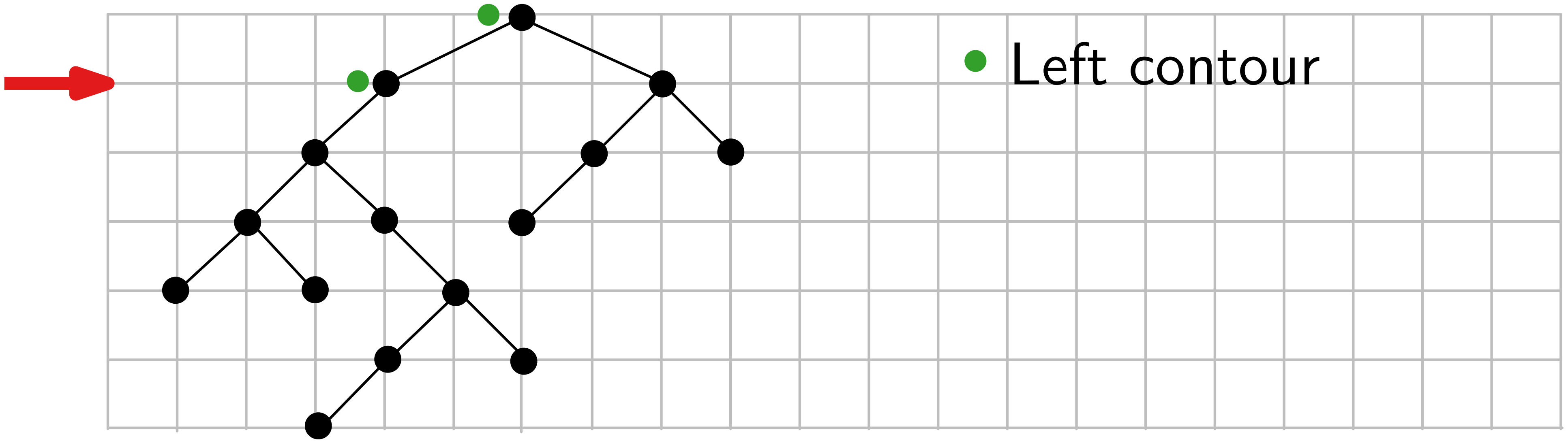
The *left/right contour* of a leveled tree drawing of height h is the sequence of vertices $(v_0, \ldots, v_h)$ such that vertex v_i is the leftmost/rightmost vertex at depth i



Implementation: Overlapping rectangles

The left/right contour of leveled tree drawing

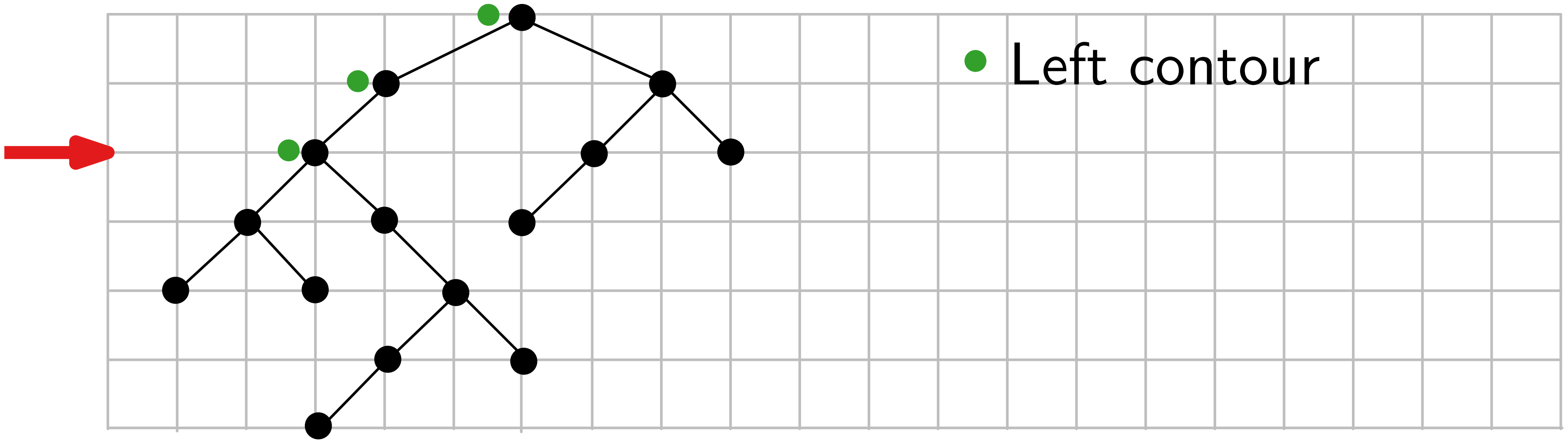
The *left/right contour* of a leveled tree drawing of height h is the sequence of vertices $(v_0, \ldots, v_h)$ such that vertex v_i is the leftmost/rightmost vertex at depth i



Implementation: Overlapping rectangles

The left/right contour of leveled tree drawing

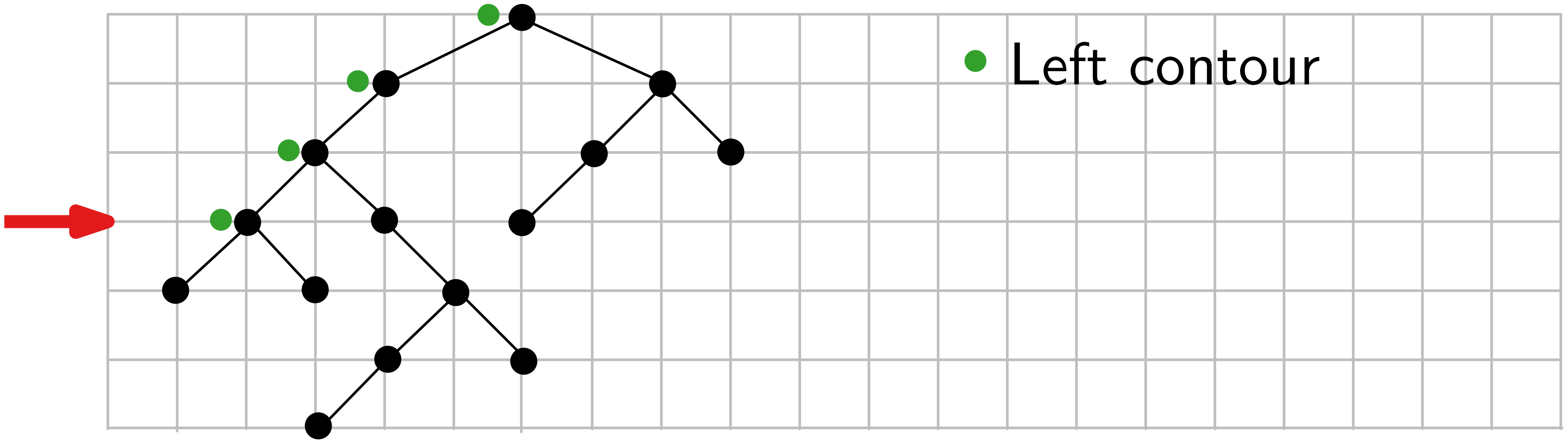
The *left/right contour* of a leveled tree drawing of height h is the sequence of vertices $(v_0, \ldots, v_h)$ such that vertex v_i is the leftmost/rightmost vertex at depth i



Implementation: Overlapping rectangles

The left/right contour of leveled tree drawing

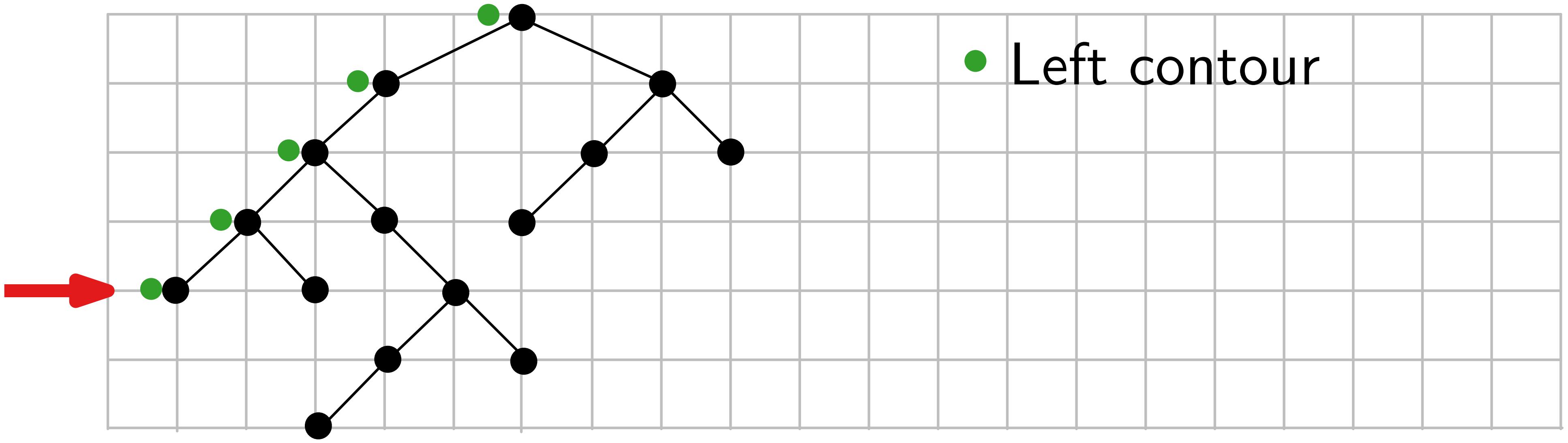
The *left/right contour* of a leveled tree drawing of height h is the sequence of vertices $(v_0, \ldots, v_h)$ such that vertex v_i is the leftmost/rightmost vertex at depth i



Implementation: Overlapping rectangles

The left/right contour of leveled tree drawing

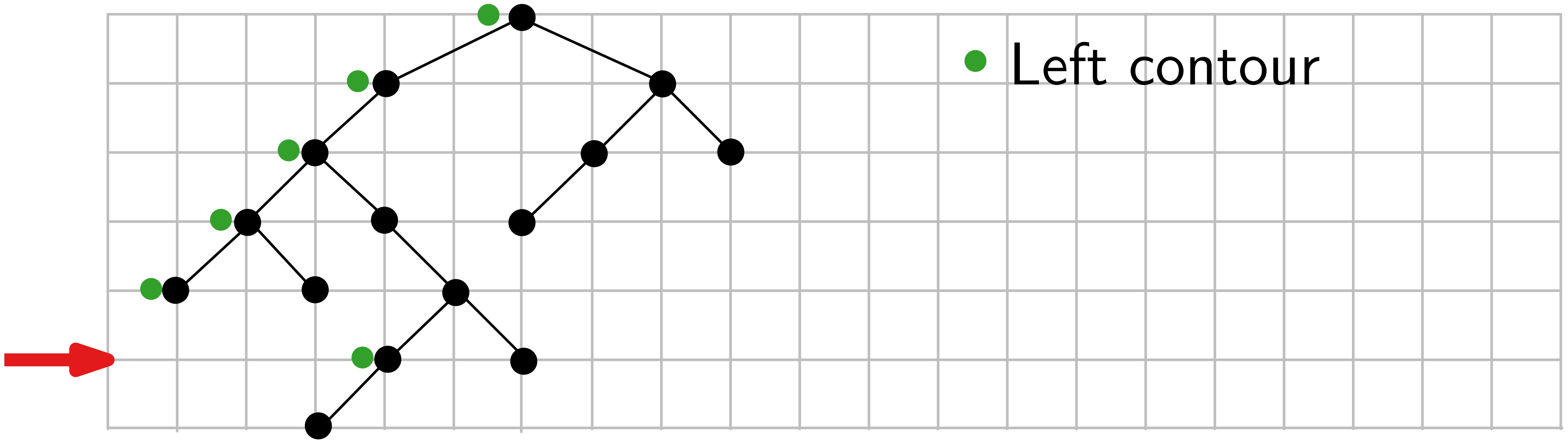
The *left/right contour* of a leveled tree drawing of height h is the sequence of vertices $(v_0, \ldots, v_h)$ such that vertex v_i is the leftmost/rightmost vertex at depth i



Implementation: Overlapping rectangles

The left/right contour of leveled tree drawing

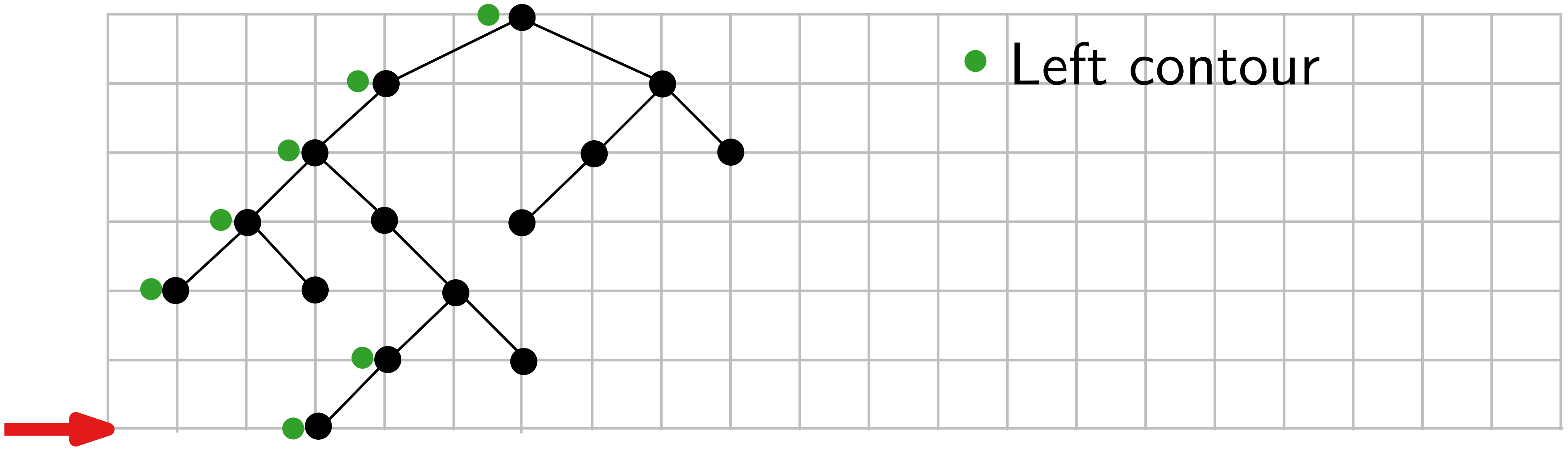
The *left/right contour* of a leveled tree drawing of height h is the sequence of vertices $(v_0, \ldots, v_h)$ such that vertex v_i is the leftmost/rightmost vertex at depth i



Implementation: Overlapping rectangles

The left/right contour of leveled tree drawing

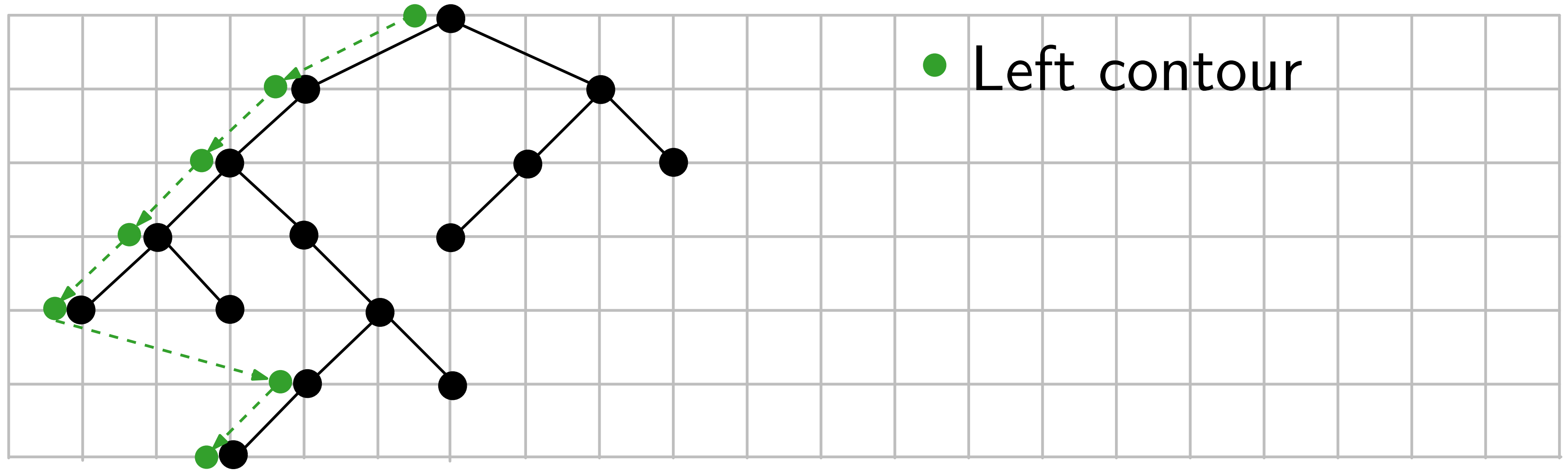
The *left/right contour* of a leveled tree drawing of height h is the sequence of vertices $(v_0, \ldots, v_h)$ such that vertex v_i is the leftmost/rightmost vertex at depth i



Implementation: Overlapping rectangles

The left/right contour of leveled tree drawing

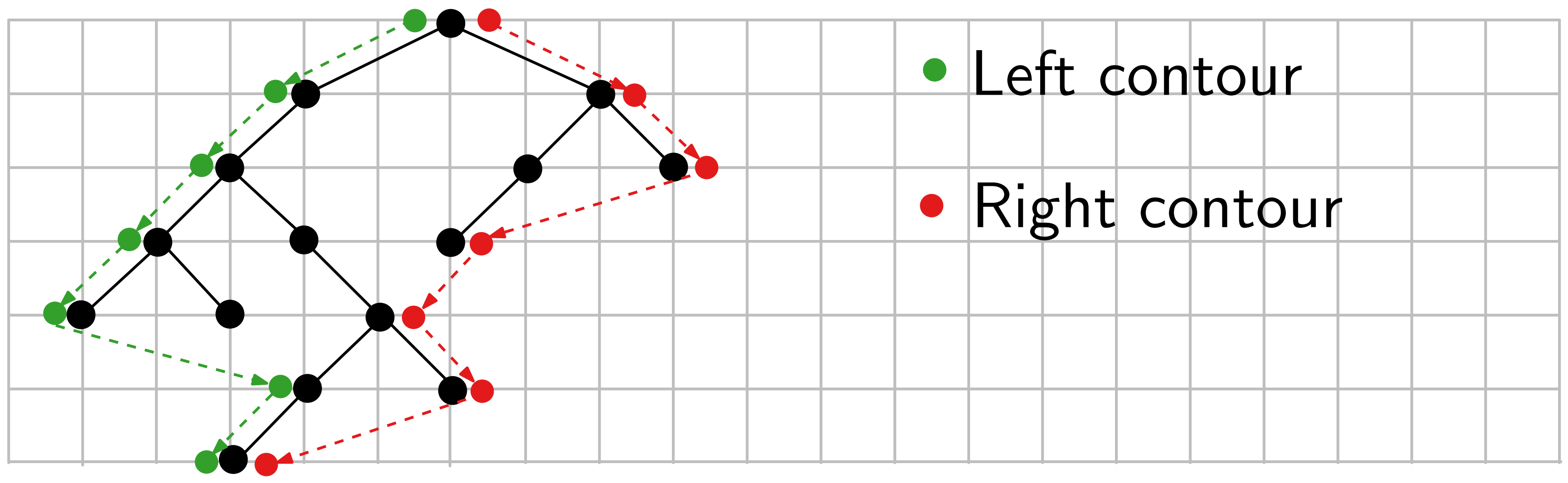
The *left/right contour* of a leveled tree drawing of height h is the sequence of vertices $(v_0, \ldots, v_h)$ such that vertex v_i is the leftmost/rightmost vertex at depth i



Implementation: Overlapping rectangles

The left/right contour of leveled tree drawing

The *left/right contour* of a leveled tree drawing of height h is the sequence of vertices $(v_0, \ldots, v_h)$ such that vertex v_i is the leftmost/rightmost vertex at depth i



Implementation: Overlapping rectangles

Computation of the left contour of a tree rooted at u, given

–the *left contours* of its subtrees

–the *heights* of its subtress

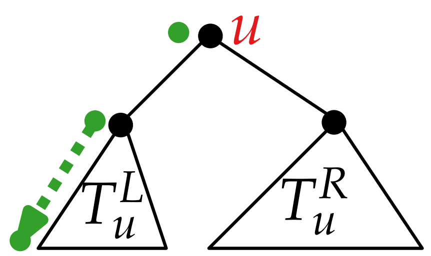
Implementation: Overlapping rectangles

Computation of the left contour of a tree rooted at u, given

–the *left contours* of its subtrees

–the *heights* of its subtress

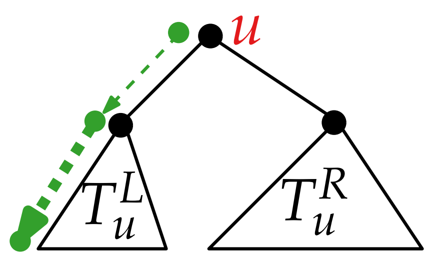
Case-1: $h(T_u^L) = h(T_u^R)$



Implementation: Overlapping rectangles

Computation of the left contour of a tree rooted at u, given
–the *left contours* of its subtrees
–the *heights* of its subtress

Case-1: $h(T_u^L) = h(T_u^R)$



$O(1)$-time

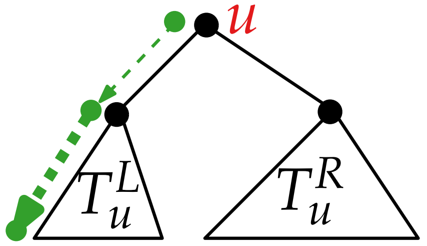
Implementation: Overlapping rectangles

Computation of the left contour of a tree rooted at u, given

–the *left contours* of its subtrees

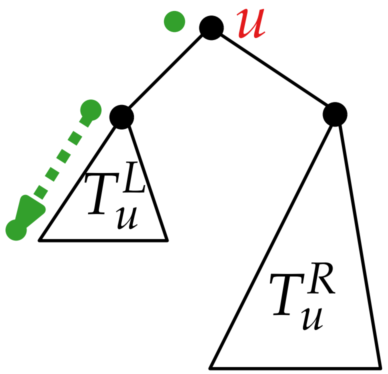
–the *heights* of its subtress

Case-1: $h(T_u^L) = h(T_u^R)$



$O(1)$-time

Case-2: $h(T_u^L) < h(T_u^R)$



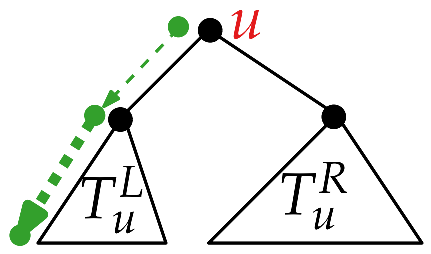
Implementation: Overlapping rectangles

Computation of the left contour of a tree rooted at u, given

–the *left contours* of its subtrees

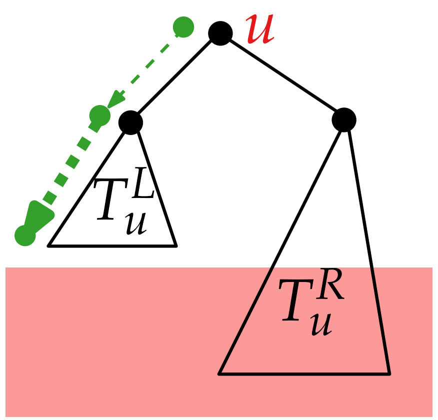
–the *heights* of its subtress

Case-1: $h(T_u^L) = h(T_u^R)$



$O(1)$-time

Case-2: $h(T_u^L) < h(T_u^R)$



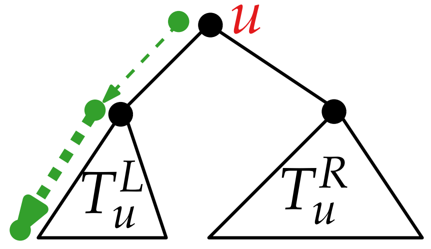
Implementation: Overlapping rectangles

Computation of the left contour of a tree rooted at u, given

–the *left contours* of its subtrees

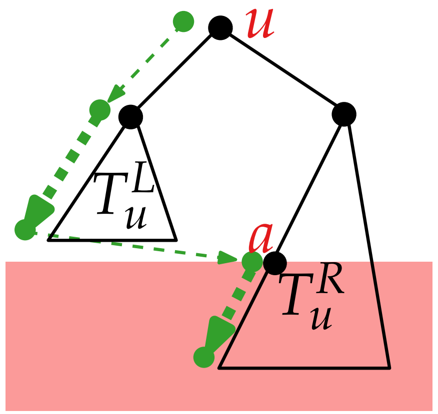
–the *heights* of its subtress

Case-1: $h(T_u^L) = h(T_u^R)$



$O(1)$-time

Case-2: $h(T_u^L) < h(T_u^R)$



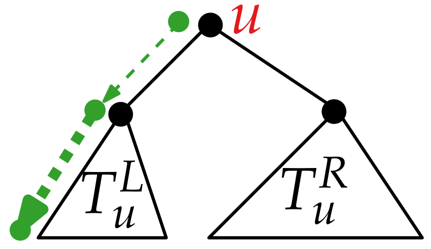
Implementation: Overlapping rectangles

Computation of the left contour of a tree rooted at u, given

–the *left contours* of its subtrees

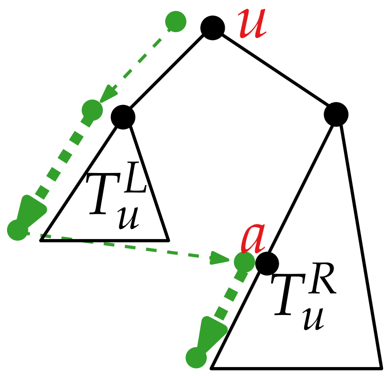
–the *heights* of its subtress

Case-1: $h(T_u^L) = h(T_u^R)$



$O(1)$-time

Case-2: $h(T_u^L) < h(T_u^R)$



$O(h(T_u^L))$-time

[We traverse T_u^L and T_u^R simultaneously in order to identify vertex a of T_u^R]

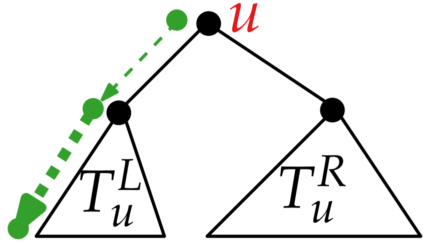
Implementation: Overlapping rectangles

Computation of the left contour of a tree rooted at u, given

–the *left contours* of its subtrees

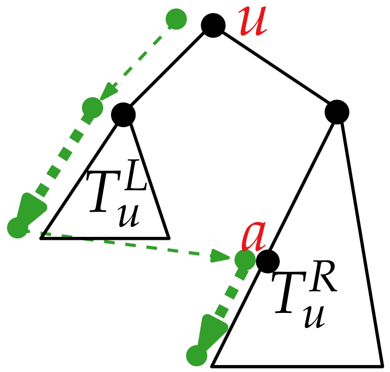
–the *heights* of its subtress

Case-1: $h(T_u^L) = h(T_u^R)$



$O(1)$-time

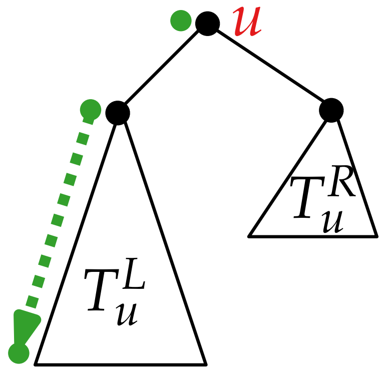
Case-2: $h(T_u^L) < h(T_u^R)$



$O(h(T_u^L))$-time

[We traverse T_u^L and T_u^R simultaneously in order to identify vertex a of T_u^R]

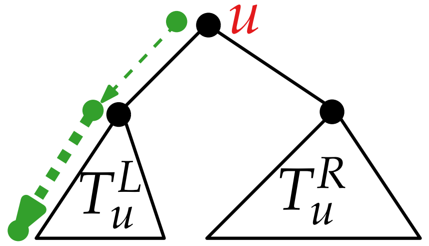
Case-3: $h(T_u^L) > h(T_u^R)$



Implementation: Overlapping rectangles

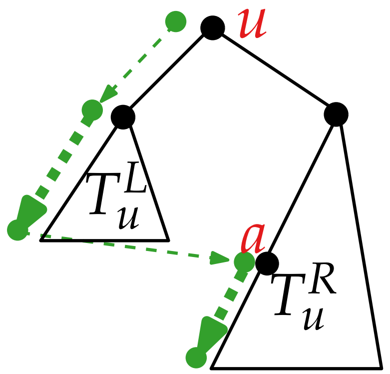
Computation of the left contour of a tree rooted at u, given
- the *left contours* of its subtrees
- the *heights* of its subtress

Case-1: $h(T_u^L) = h(T_u^R)$



$O(1)$-time

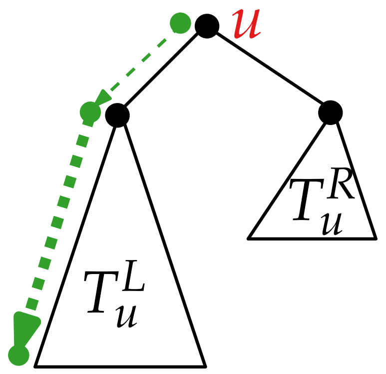
Case-2: $h(T_u^L) < h(T_u^R)$



$O(h(T_u^L))$-time

[We traverse T_u^L and T_u^R simultaneously in order to identify vertex a of T_u^R]

Case-3: $h(T_u^L) > h(T_u^R)$



$O(1)$-time

Implementation: Overlapping rectangles

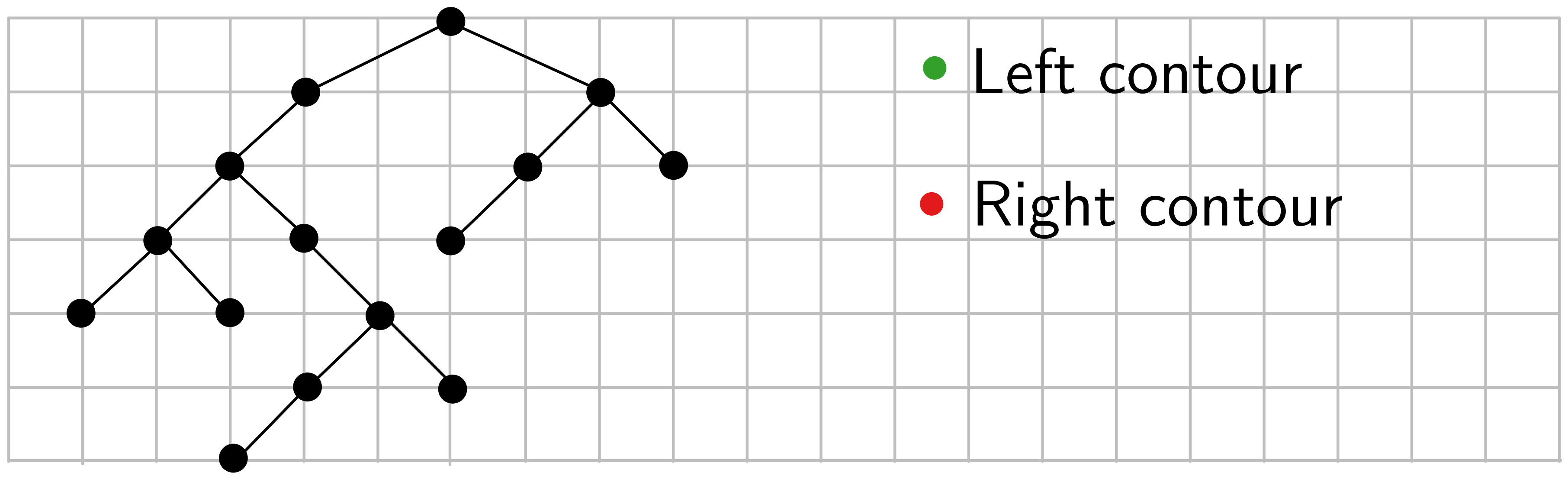
Total cost for computing the contours of a tree:

[We build each contour in a bottom-up fashion through a postorder traversal.]

Implementation: Overlapping rectangles

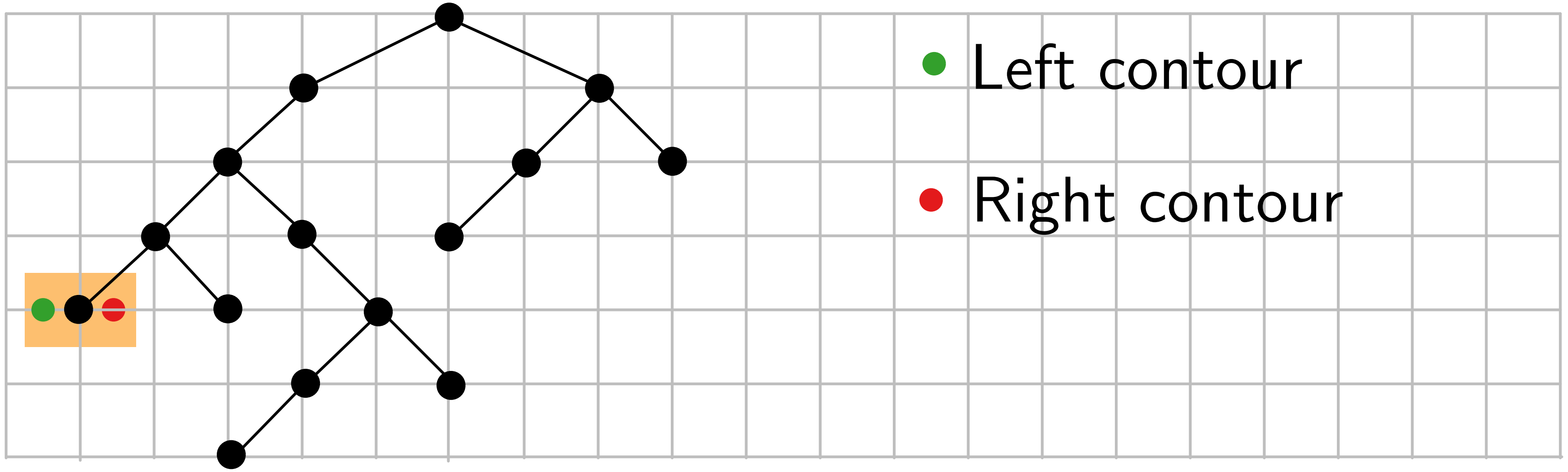
Total cost for computing the contours of a tree:

[We build each contour in a bottom-up fashion through a postorder traversal.]



Implementation: Overlapping rectangles

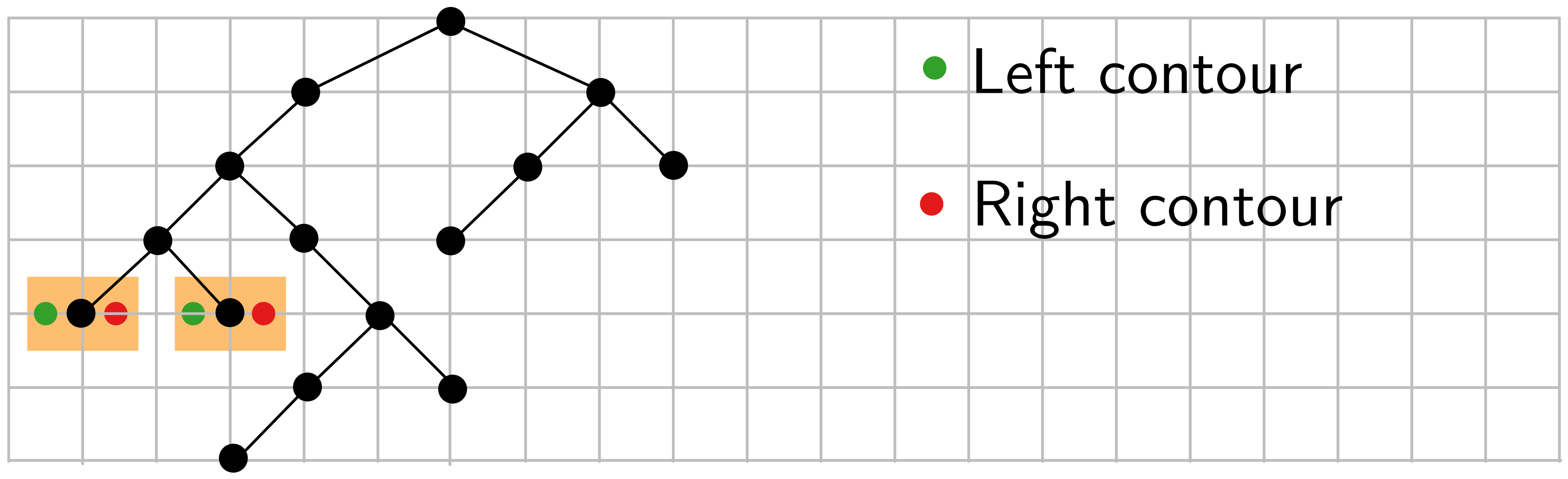
Total cost for computing the contours of a tree:
[We build each contour in a bottom-up fashion through a postorder traversal.]



Implementation: Overlapping rectangles

Total cost for computing the contours of a tree:

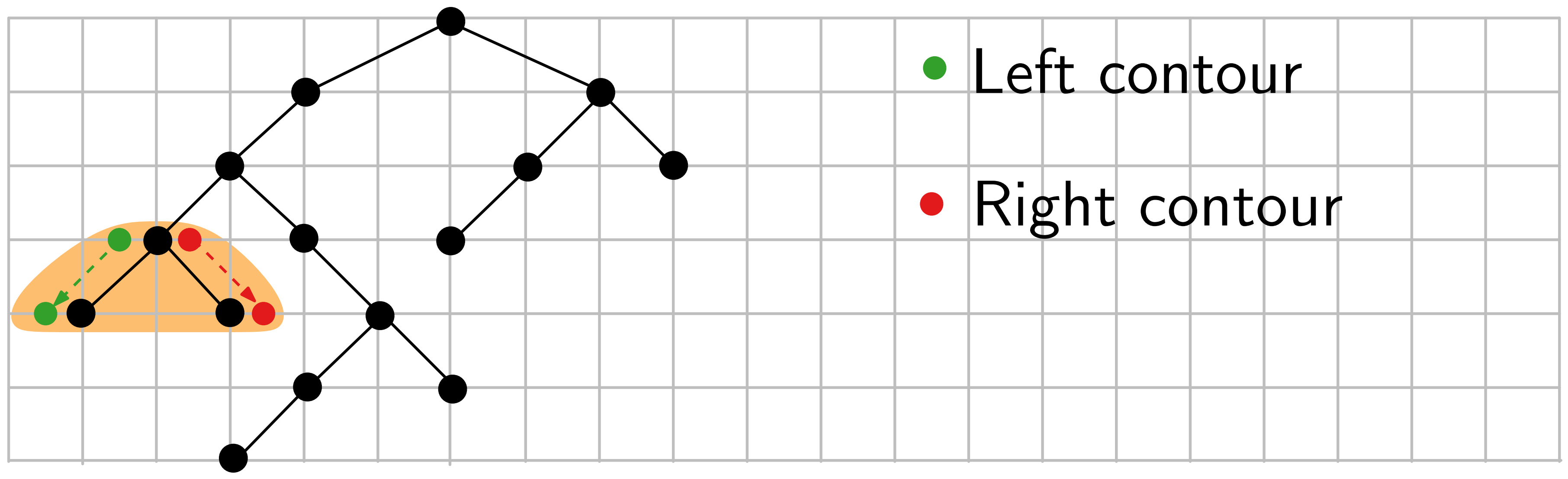
[We build each contour in a bottom-up fashion through a postorder traversal.]



Implementation: Overlapping rectangles

Total cost for computing the contours of a tree:

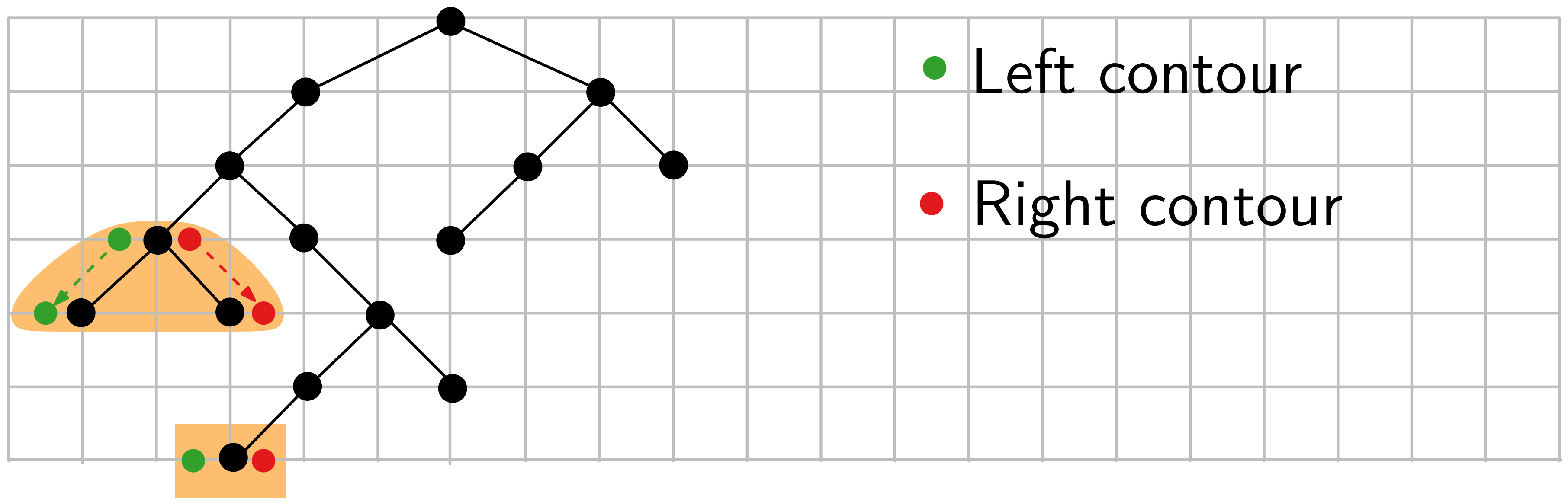
[We build each contour in a bottom-up fashion through a postorder traversal.]



Implementation: Overlapping rectangles

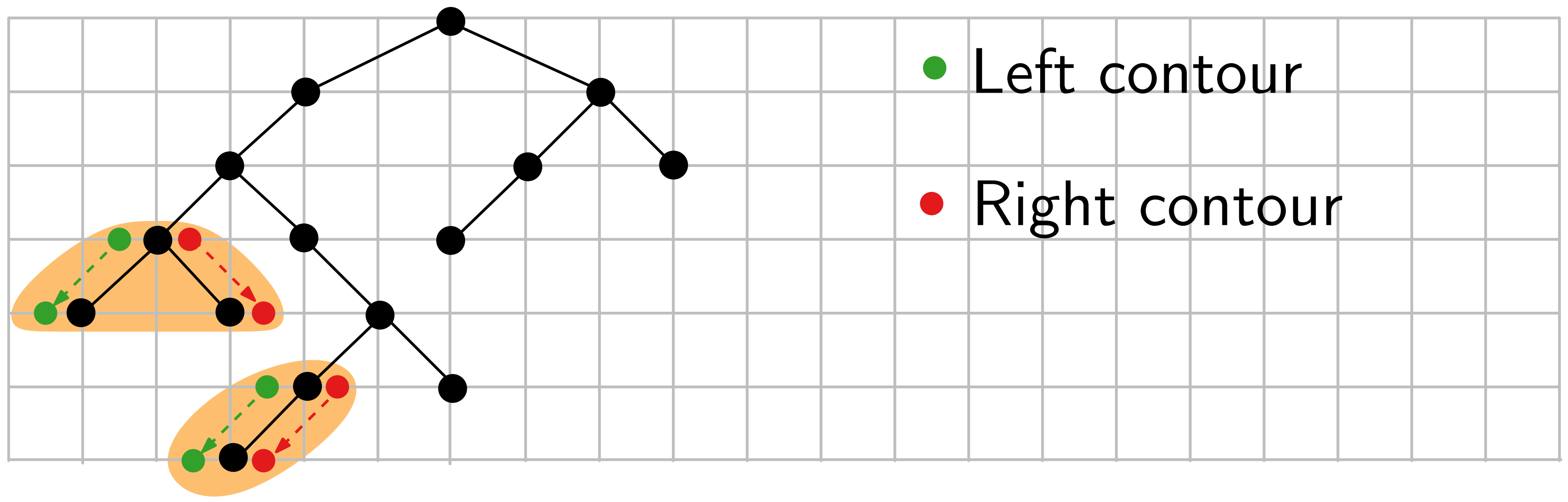
Total cost for computing the contours of a tree:

[We build each contour in a bottom-up fashion through a postorder traversal.]



Implementation: Overlapping rectangles

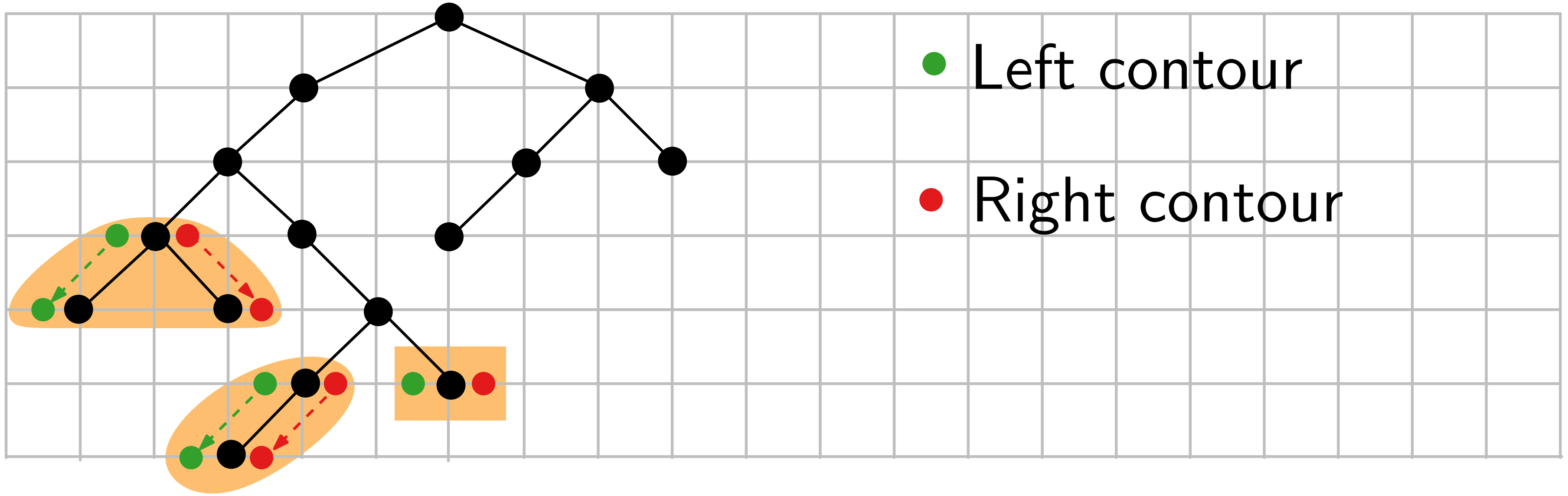
Total cost for computing the contours of a tree:
[We build each contour in a bottom-up fashion through a postorder traversal.]



Implementation: Overlapping rectangles

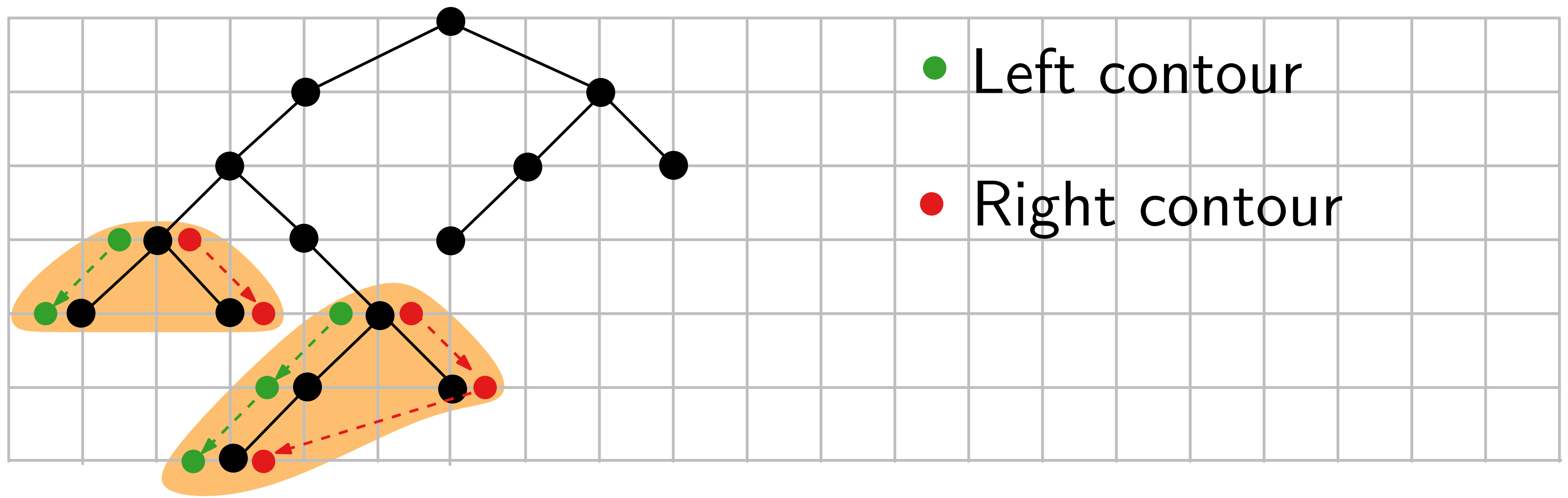
Total cost for computing the contours of a tree:

[We build each contour in a bottom-up fashion through a postorder traversal.]



Implementation: Overlapping rectangles

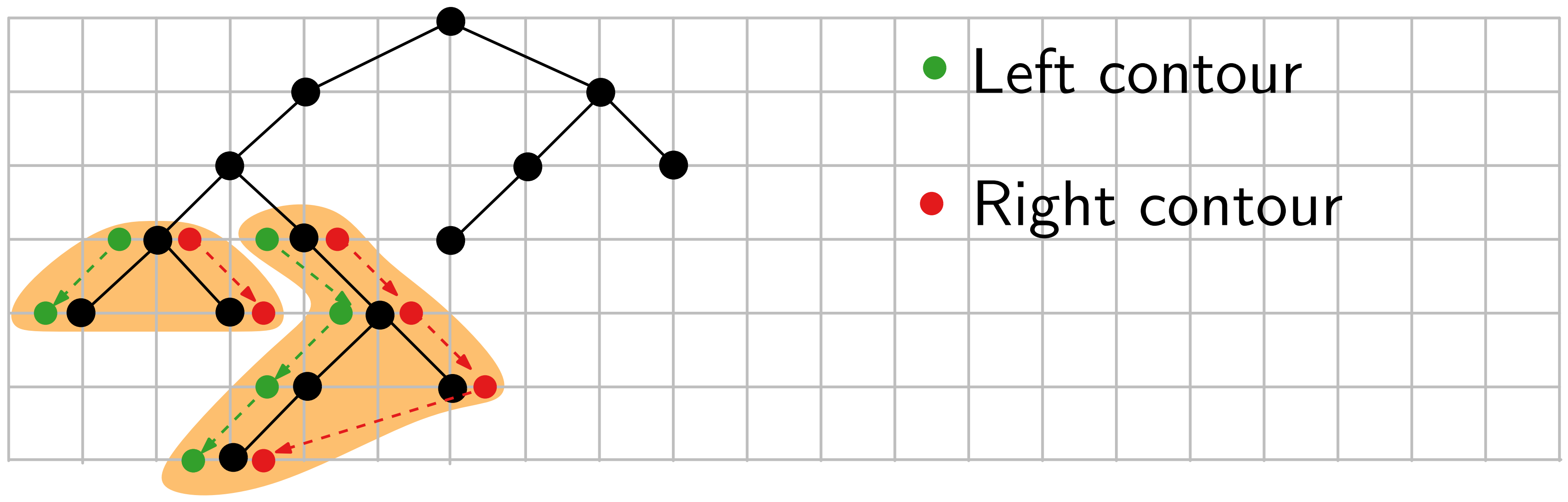
Total cost for computing the contours of a tree:
[We build each contour in a bottom-up fashion through a postorder traversal.]



Implementation: Overlapping rectangles

Total cost for computing the contours of a tree:

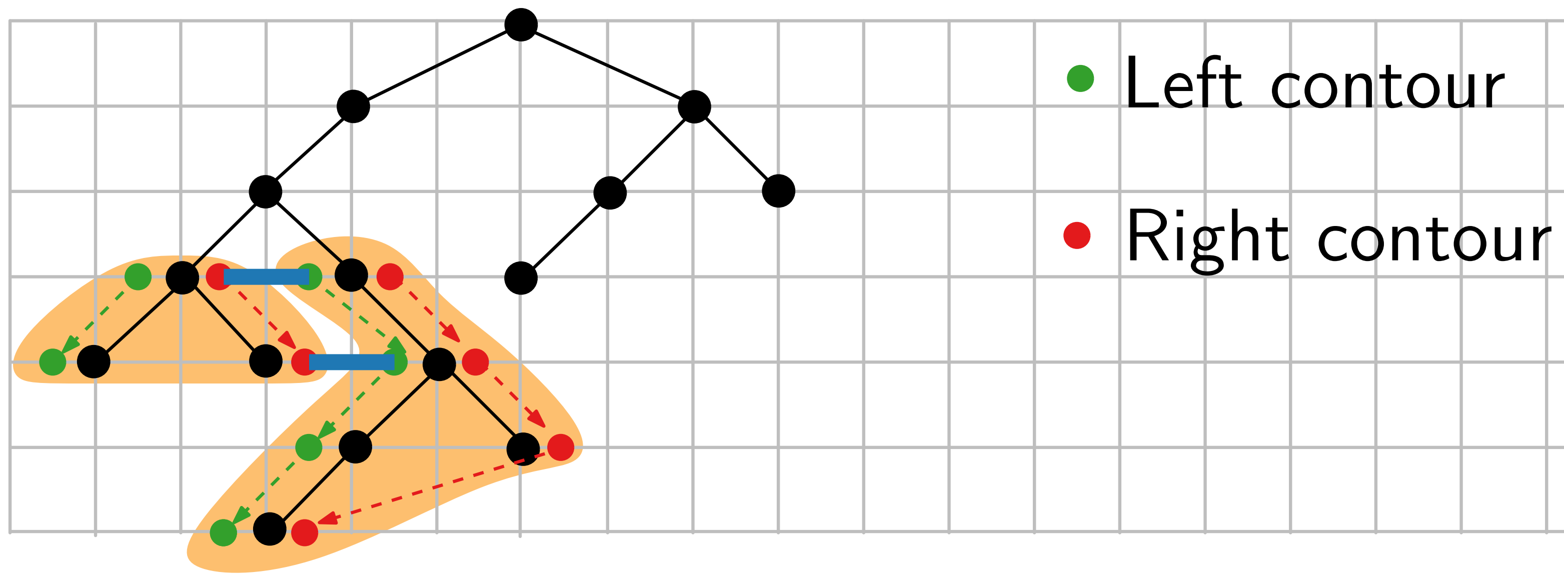
[We build each contour in a bottom-up fashion through a postorder traversal.]



Implementation: Overlapping rectangles

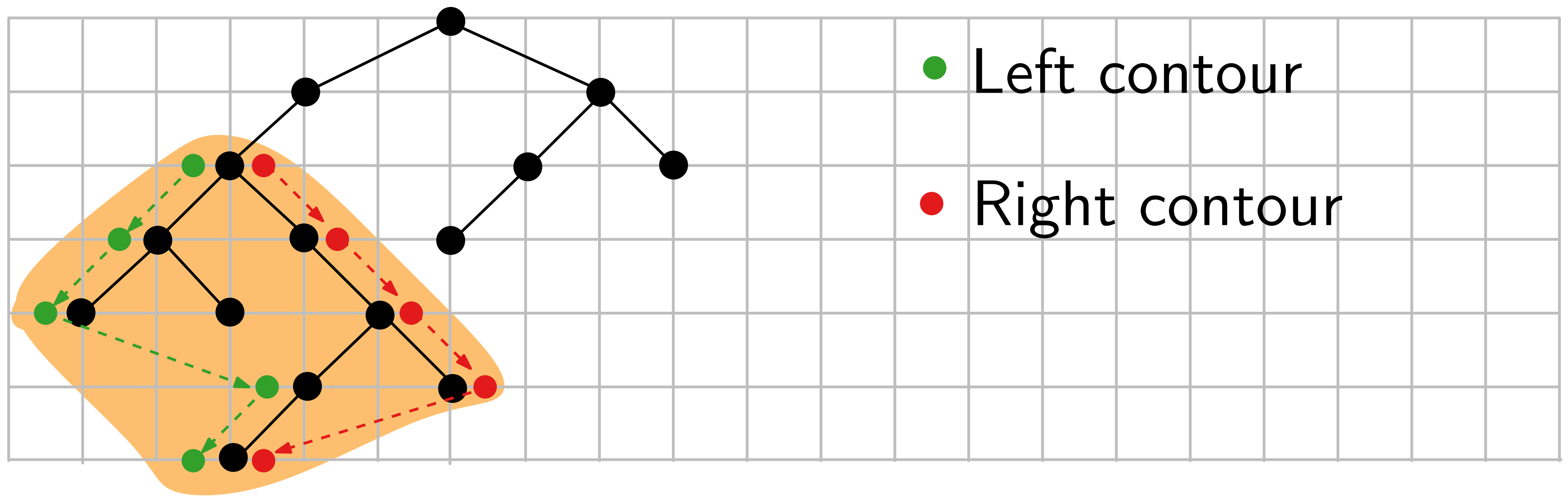
Total cost for computing the contours of a tree:

[We build each contour in a bottom-up fashion through a postorder traversal.]



Implementation: Overlapping rectangles

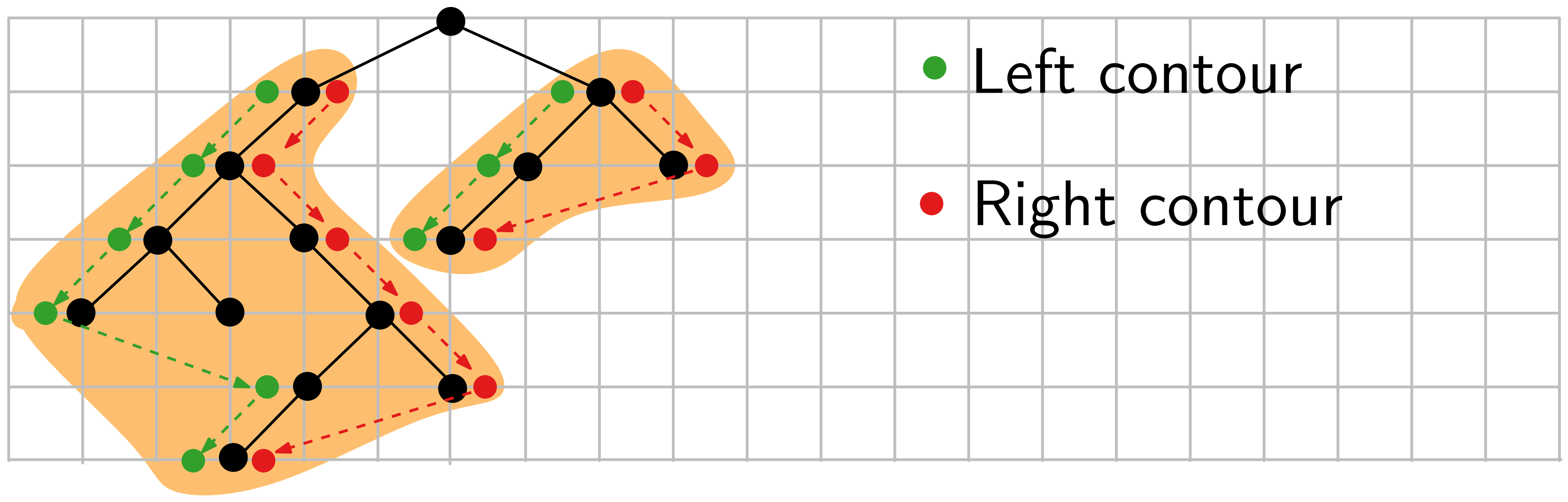
Total cost for computing the contours of a tree:
[We build each contour in a bottom-up fashion through a postorder traversal.]



Implementation: Overlapping rectangles

Total cost for computing the contours of a tree:

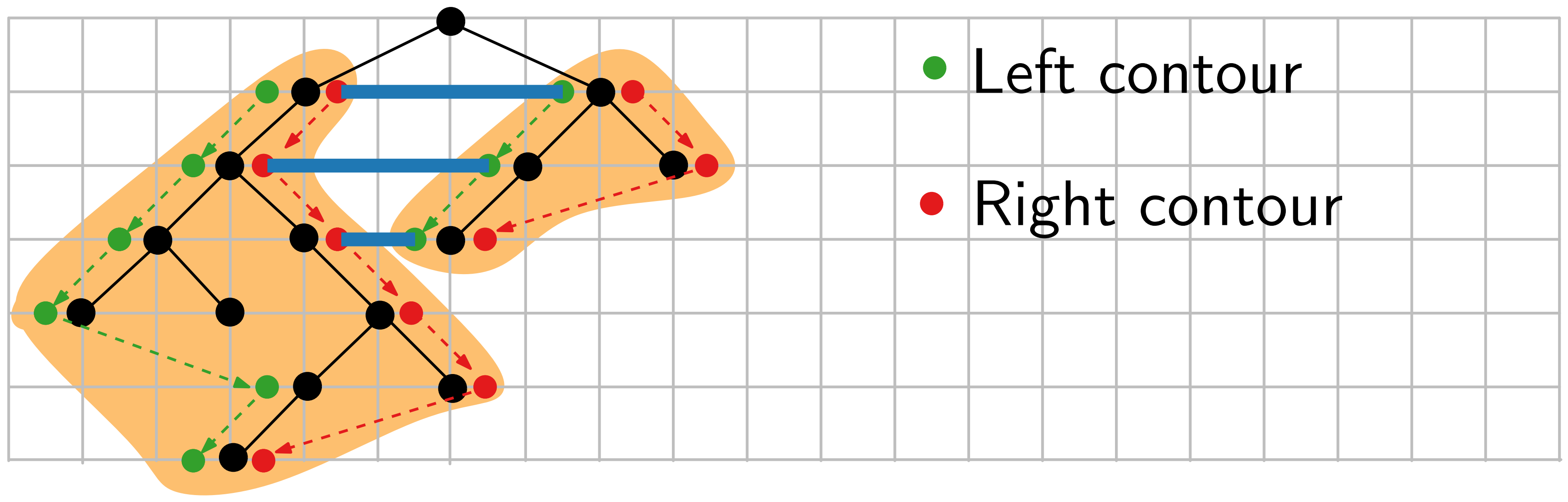
[We build each contour in a bottom-up fashion through a postorder traversal.]



Implementation: Overlapping rectangles

Total cost for computing the contours of a tree:

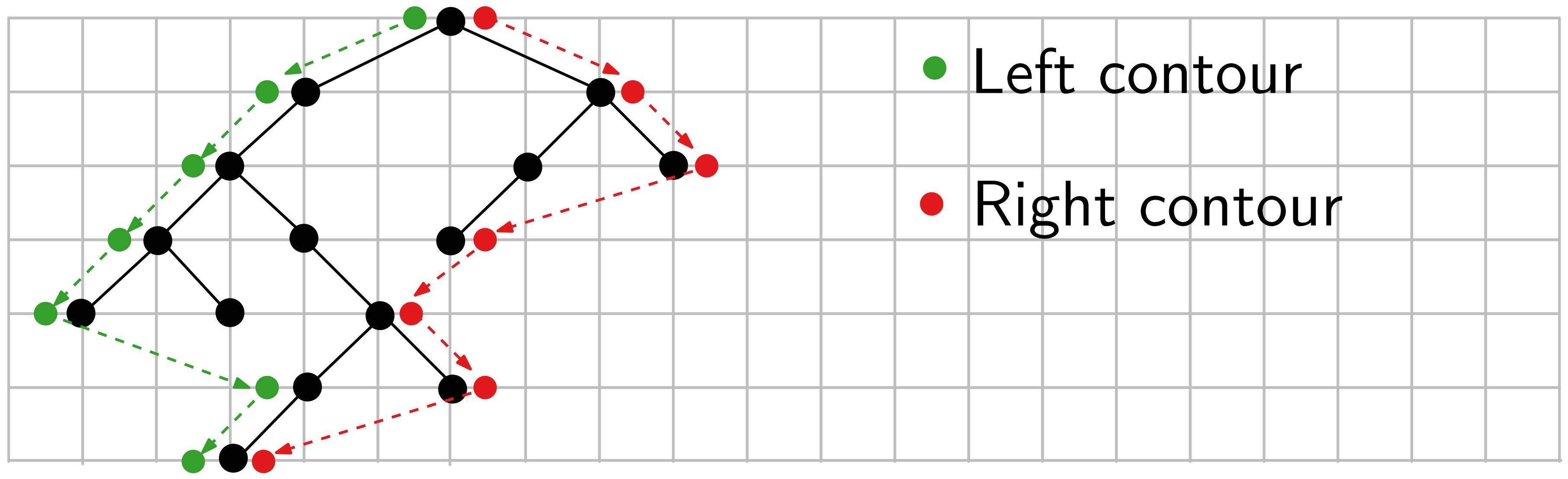
[We build each contour in a bottom-up fashion through a postorder traversal.]



Implementation: Overlapping rectangles

Total cost for computing the contours of a tree:

[We build each contour in a bottom-up fashion through a postorder traversal.]



Implementation: Overlapping rectangles

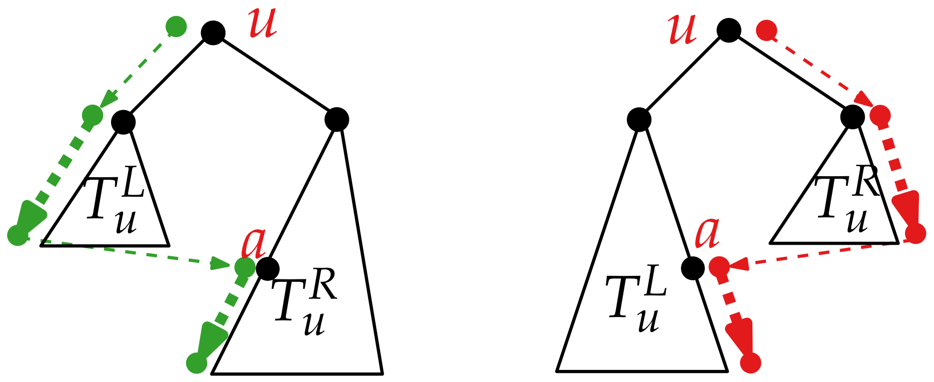
Total cost for computing the contours of a tree:
[We build each contour in a bottom-up fashion through a postorder traversal.]

Implementation: Overlapping rectangles

Total cost for computing the contours of a tree:

[We build each contour in a bottom-up fashion through a postorder traversal.]

$$C(T) \leq \sum_{u \in V(T)} 1 + \min(h(T_u^L), h(T_u^R))$$

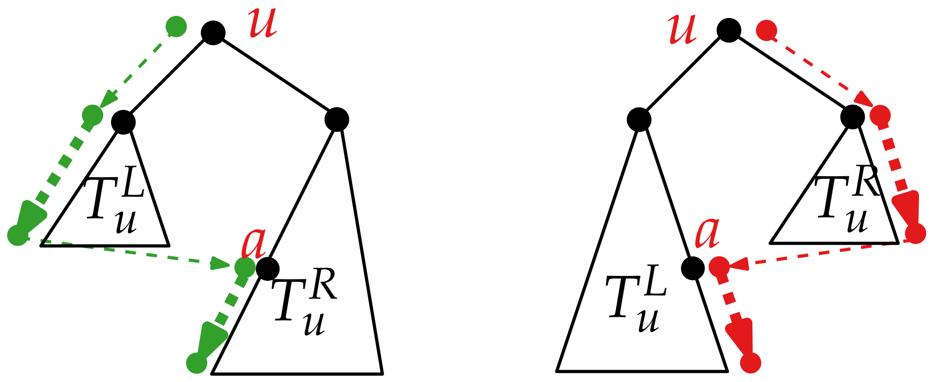


Implementation: Overlapping rectangles

Total cost for computing the contours of a tree:

[We build each contour in a bottom-up fashion through a postorder traversal.]

$$
\begin{aligned}
C(T) &\leq \sum_{u \in V(T)} 1 + \min(h(T_u^L), h(T_u^R)) \\
&= n + \sum_{u \in V(T)} \min(h(T_u^L), h(T_u^R)) \\
&< n + n \qquad \textbf{(Lemma 1)} \\
&= 2n
\end{aligned}
$$



Thus, $C(T) \leq 2n$

Implementation: Overlapping rectangles

Lemma 1: For each n-vertex binary tree it holds that:

$$\sum_{u \in V(T)} \min(h(T_u^L), h(T_u^R)) < n$$

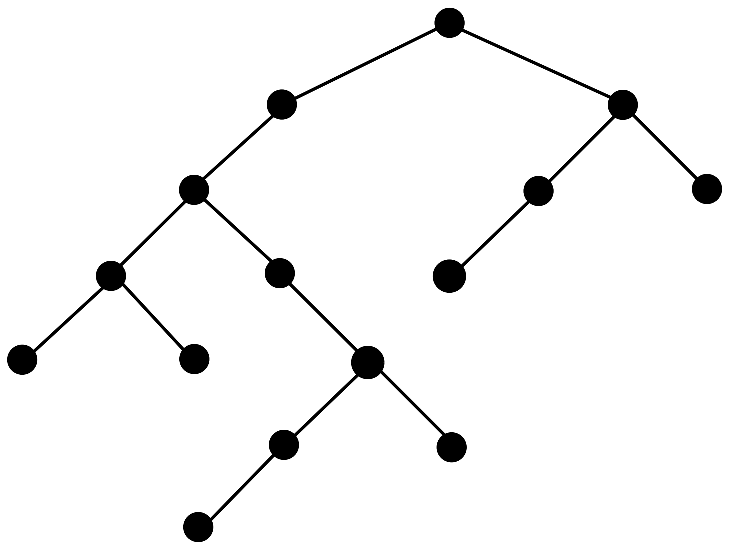
Implementation: Overlapping rectangles

Lemma 1: For each n-vertex binary tree it holds that:

$$\sum_{u \in V(T)} \min(h(T_u^L), h(T_u^R)) < n$$

Proof:

- The height of each subtree is equal to the length of the left/right contour
- We connect each vertex from contour of the shorter subtree to the visible vertex on the contour of the opposite subtree.



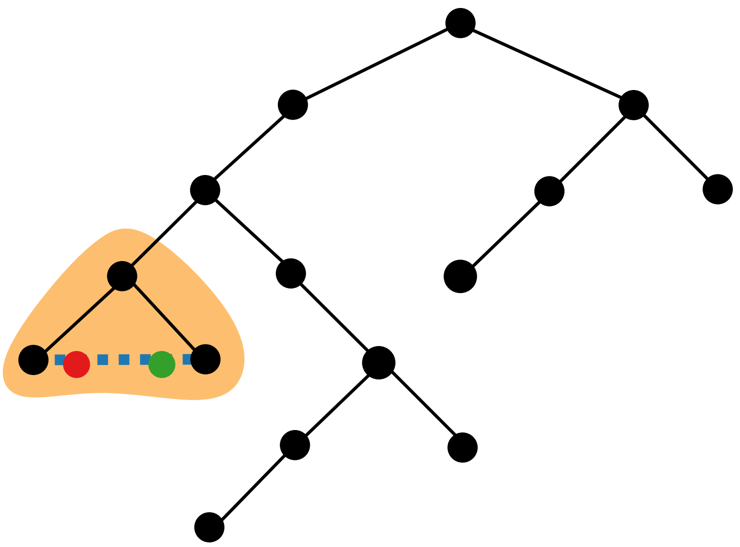
Implementation: Overlapping rectangles

Lemma 1: For each n-vertex binary tree it holds that:

$$\sum_{u \in V(T)} \min(h(T_u^L), h(T_u^R)) < n$$

Proof:

- The height of each subtree is equal to the length of the left/right contour
- We connect each vertex from contour of the shorter subtree to the visible vertex on the contour of the opposite subtree.



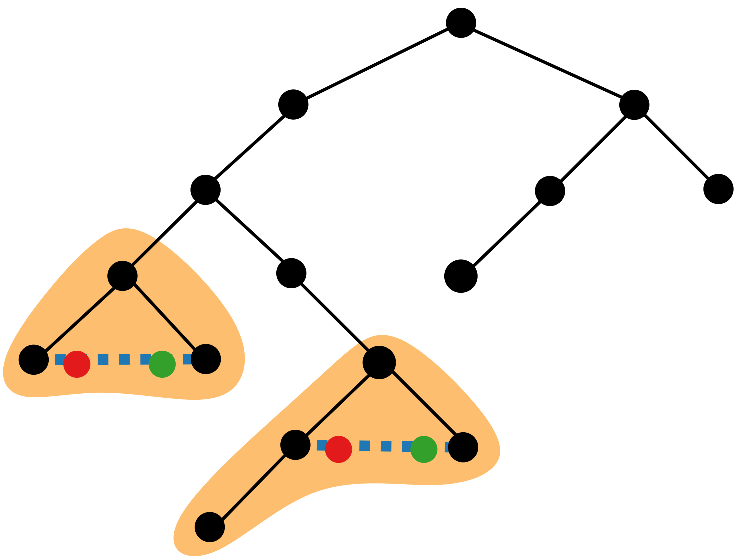
Implementation: Overlapping rectangles

Lemma 1: For each n-vertex binary tree it holds that:

$$\sum_{u \in V(T)} \min(h(T_u^L), h(T_u^R)) < n$$

Proof:

- The height of each subtree is equal to the length of the left/right contour
- We connect each vertex from contour of the shorter subtree to the visible vertex on the contour of the opposite subtree.



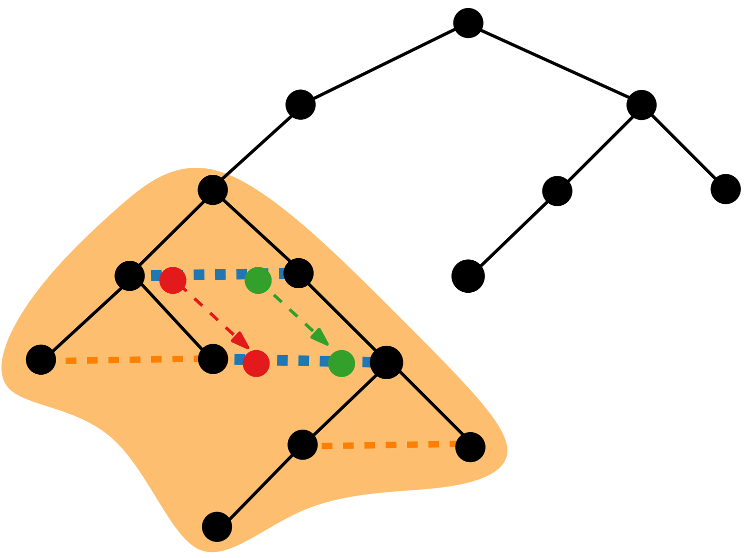
Implementation: Overlapping rectangles

Lemma 1: For each n-vertex binary tree it holds that:

$$\sum_{u \in V(T)} \min(h(T_u^L), h(T_u^R)) < n$$

Proof:

- The height of each subtree is equal to the length of the left/right contour
- We connect each vertex from contour of the shorter subtree to the visible vertex on the contour of the opposite subtree.



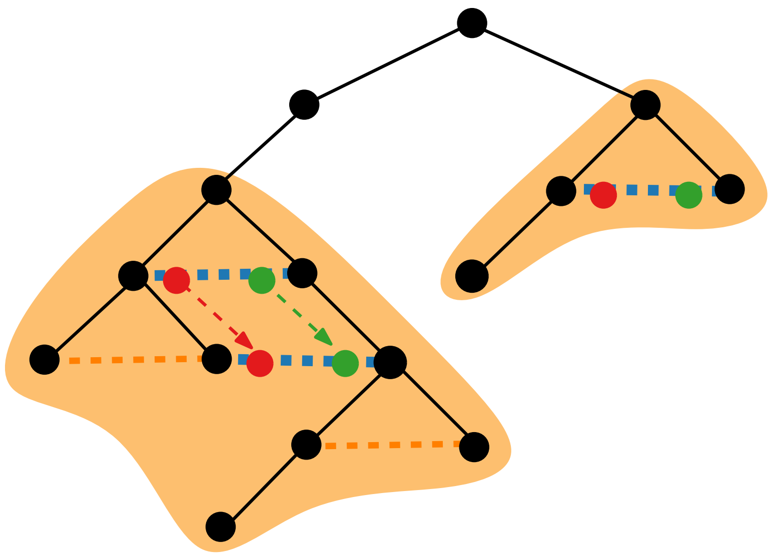
Implementation: Overlapping rectangles

Lemma 1: For each n-vertex binary tree it holds that:

$$\sum_{u \in V(T)} \min(h(T_u^L), h(T_u^R)) < n$$

Proof:

- The height of each subtree is equal to the length of the left/right contour
- We connect each vertex from contour of the shorter subtree to the visible vertex on the contour of the opposite subtree.



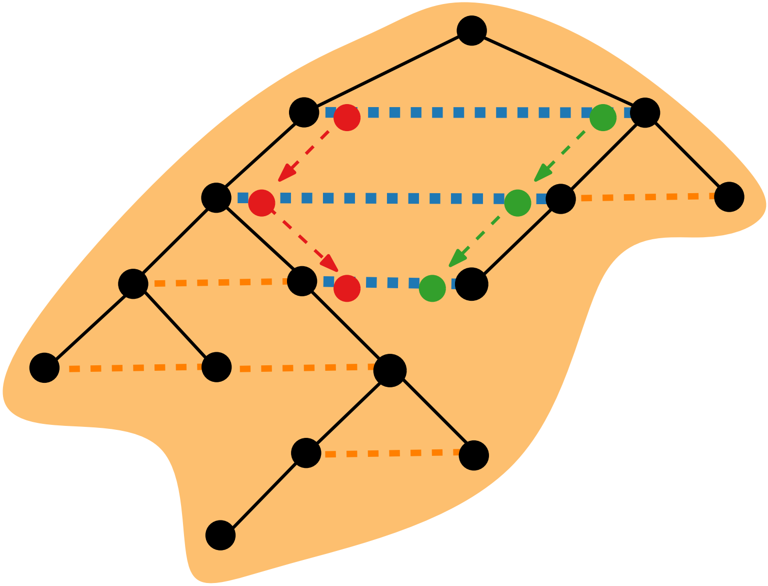
Implementation: Overlapping rectangles

Lemma 1: For each n-vertex binary tree it holds that:

$$\sum_{u \in V(T)} \min(h(T_u^L), h(T_u^R)) < n$$

Proof:

- The height of each subtree is equal to the length of the left/right contour
- We connect each vertex from contour of the shorter subtree to the visible vertex on the contour of the opposite subtree.



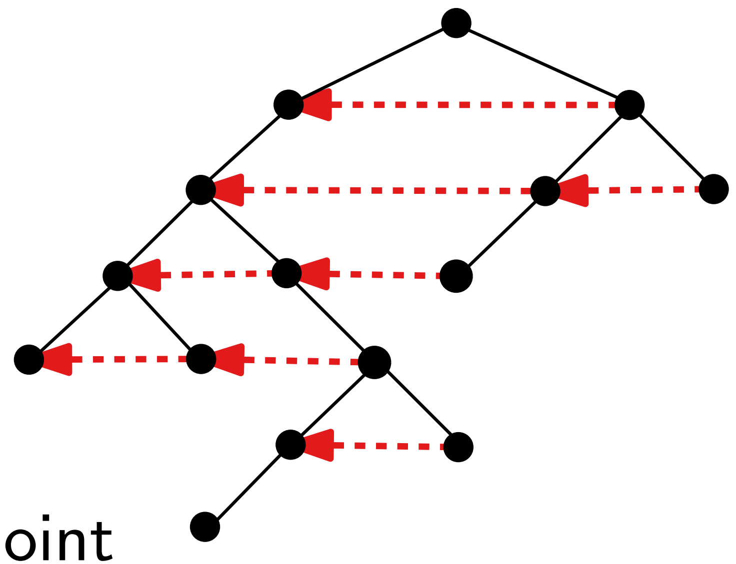
Implementation: Overlapping rectangles

Lemma 1: For each n-vertex binary tree it holds that:

$$\sum_{u \in V(T)} \min(h(T_u^L), h(T_u^R)) < n$$

Proof:

- The height of each subtree is equal to the length of the left/right contour
- We connect each vertex from contour of the shorter subtree to the visible vertex on the contour of the opposite subtree.
- We can charge each connection to the vertex at its left endpoint
- Observe that we have at most one connection out of the right side of each vertex. Thus, at most n connections. $\square$



Level-based layout – result

Theorem. (Reingold & Tilford '81)
Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time, such that:

Level-based layout – result

Theorem. (Reingold & Tilford '81)
Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time, such that:

- Γ is planar, straight-line and strictly downward
- Γ is leveled: y-coordinate of vertex v is $-\text{depth}(v)$
- Vertical and horizontal distances are at least 1
- Each vertex is centred wrt its children

Level-based layout – result

Theorem. (Reingold & Tilford ’81)
Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time, such that:

- Γ is planar, straight-line and strictly downward
- Γ is leveled: y-coordinate of vertex v is $-\text{depth}(v)$
- Vertical and horizontal distances are at least 1
- Each vertex is centred wrt its children
- Area of Γ is in $\mathcal{O}(n^2)$

Level-based layout – result

Theorem. (Reingold & Tilford ’81)
Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time, such that:

- Γ is planar, straight-line and strictly downward
- Γ is leveled: y-coordinate of vertex v is $-\mathrm{depth}(v)$
- Vertical and horizontal distances are at least 1
- Each vertex is centred wrt its children
- Area of Γ is in $\mathcal{O}(n^2)$
- Simply isomorphic subtrees have congruent drawings, up to translation
- Axially isomorphic trees have congruent drawings, up to translation and reflection around y-axis

Level-based layout – result

generalisable

Theorem. (Reingold & Tilford '81)
Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time, such that:

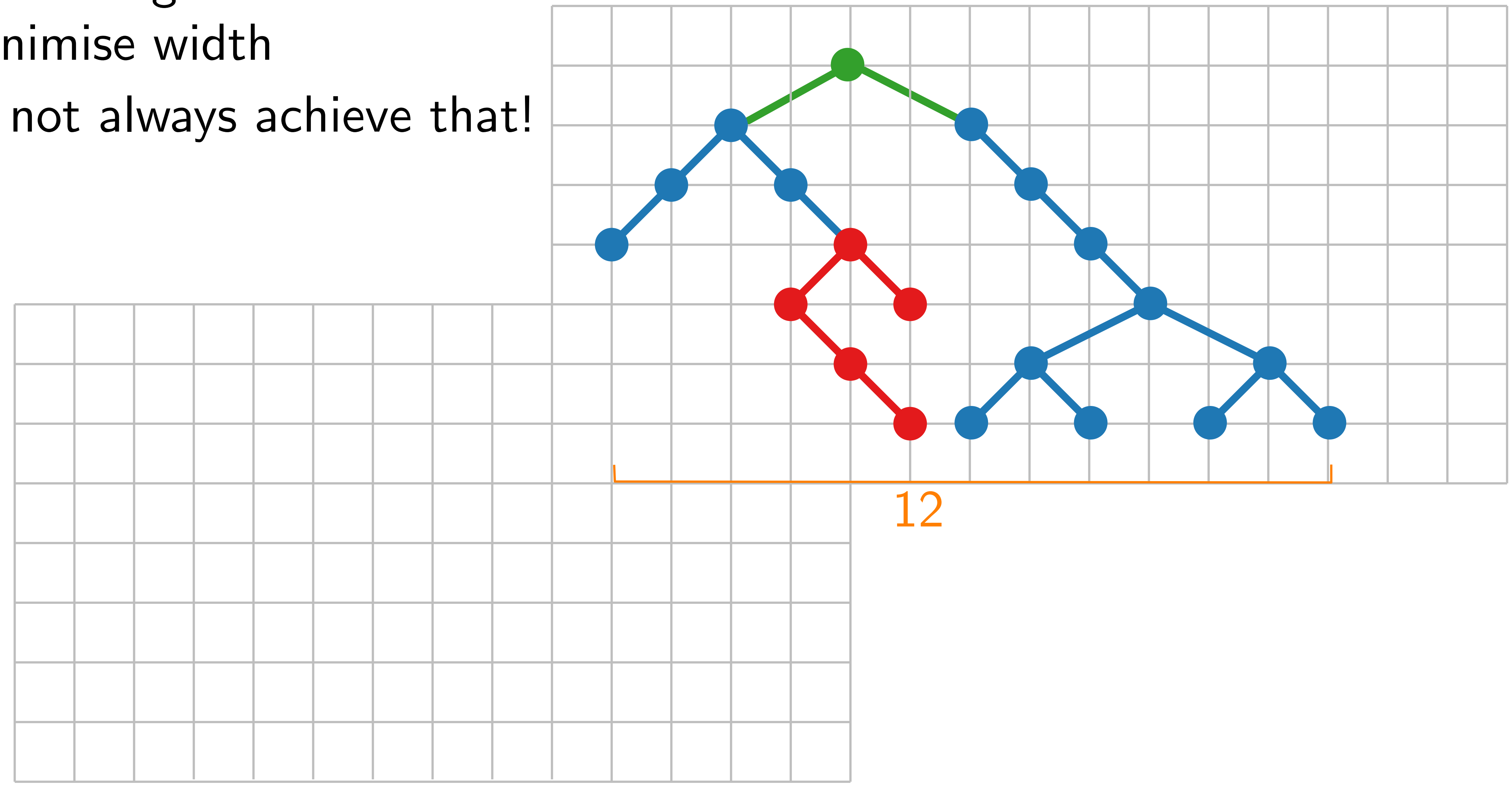
- Γ is planar, straight-line and strictly downward
- Γ is leveled: y-coordinate of vertex v is $-\text{depth}(v)$
- Vertical and horizontal distances are at least 1
- Each vertex is centred wrt its children
- Area of Γ is in $\mathcal{O}(n^2)$
- Simply isomorphic subtrees have congruent drawings, up to translation
- Axially isomorphic trees have congruent drawings, up to translation and reflection around y-axis

Level-based layout – area

- Presented algorithm tries to minimise width
- Does not always achieve that!

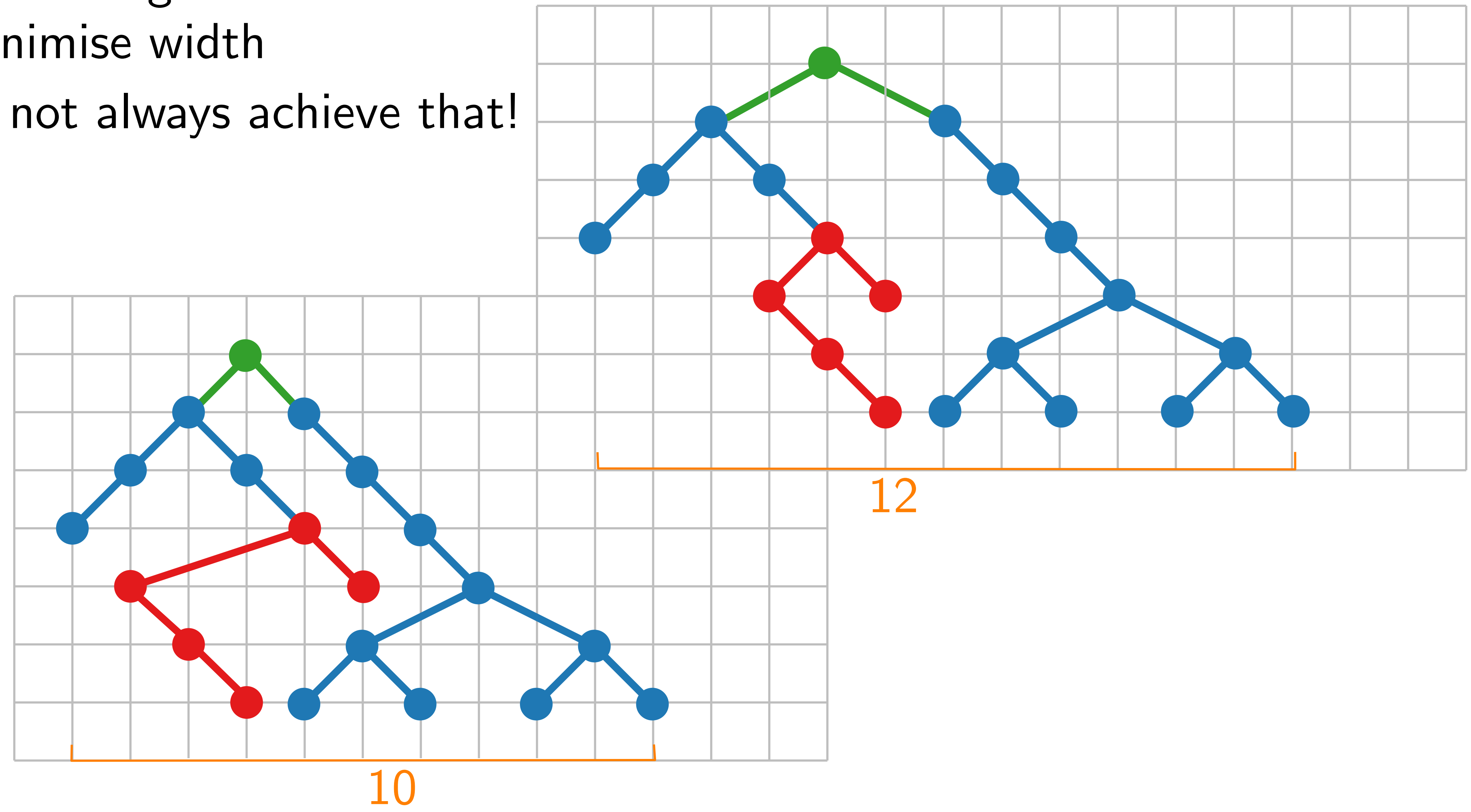
Level-based layout – area

- Presented algorithm tries to minimise width
- Does not always achieve that!



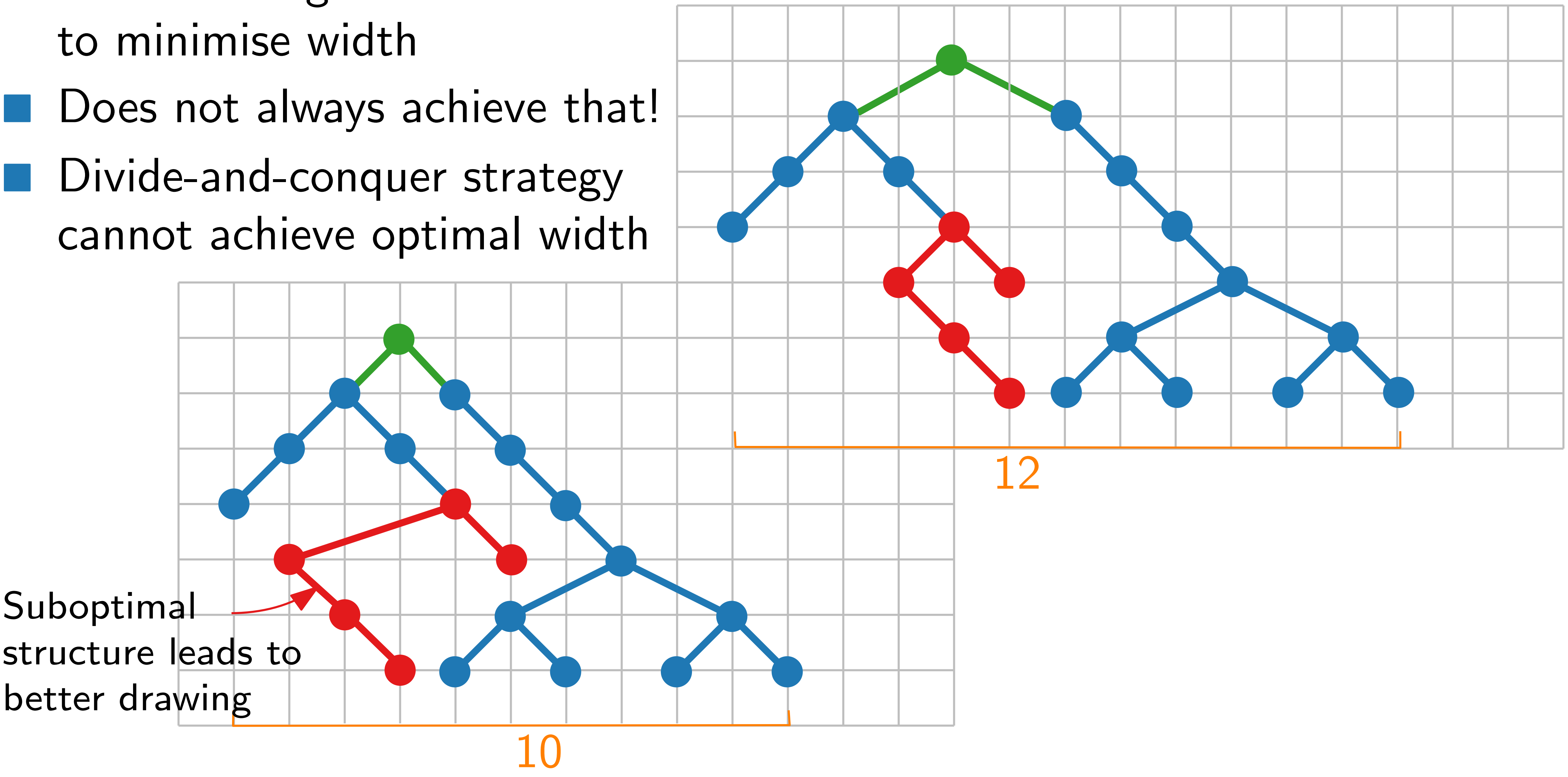
Level-based layout – area

- Presented algorithm tries to minimise width
- Does not always achieve that!



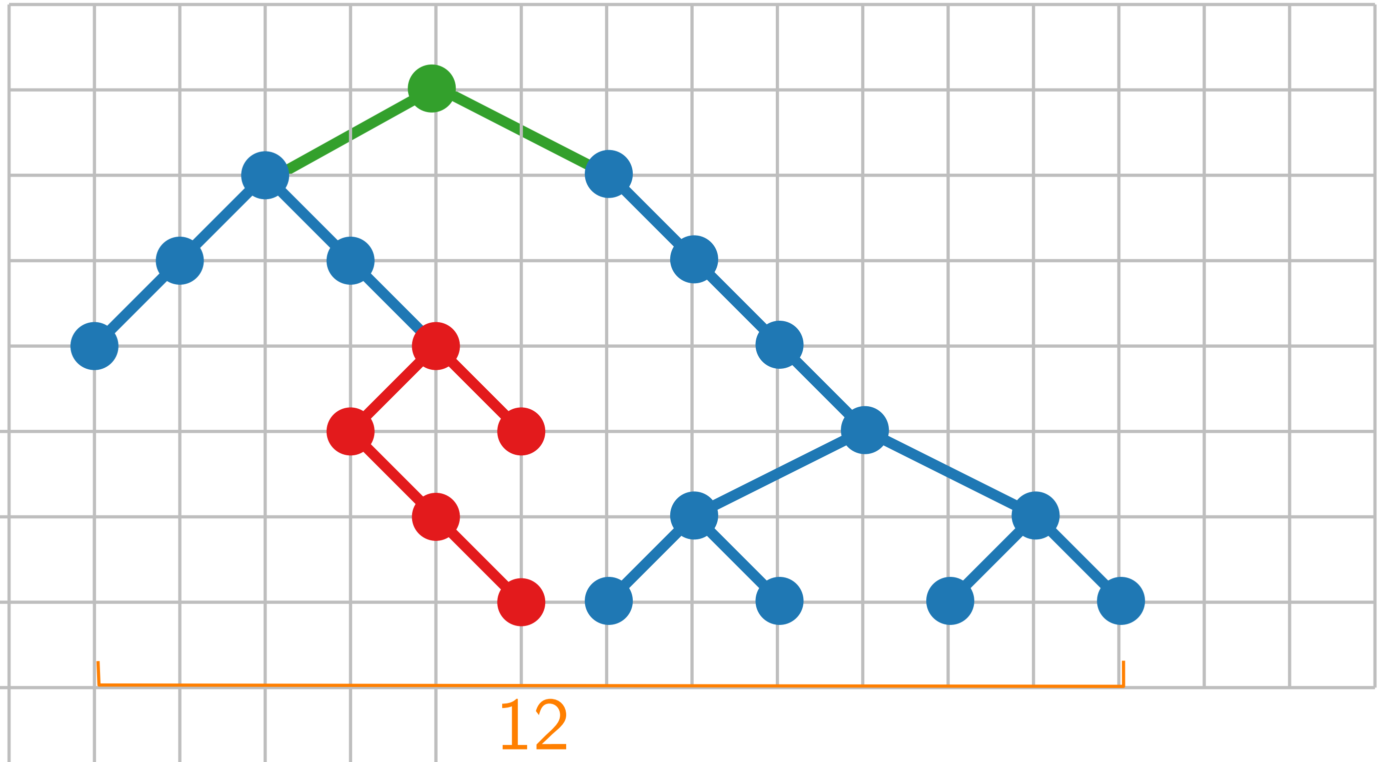
Level-based layout – area

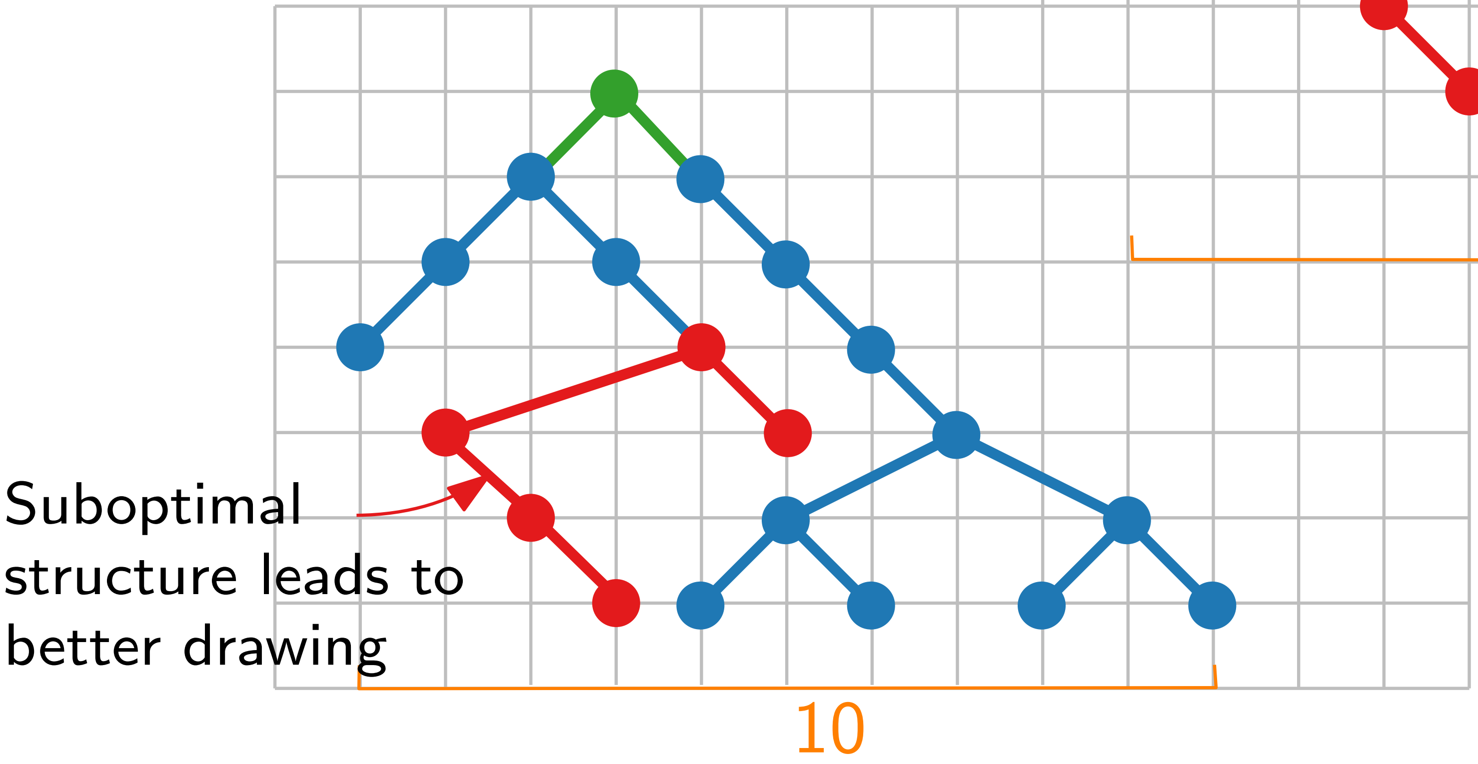
- Presented algorithm tries to minimise width
- Does not always achieve that!
- Divide-and-conquer strategy cannot achieve optimal width



Level-based layout – area

- Presented algorithm tries to minimise width
- Does not always achieve that!
- Divide-and-conquer strategy cannot achieve optimal width

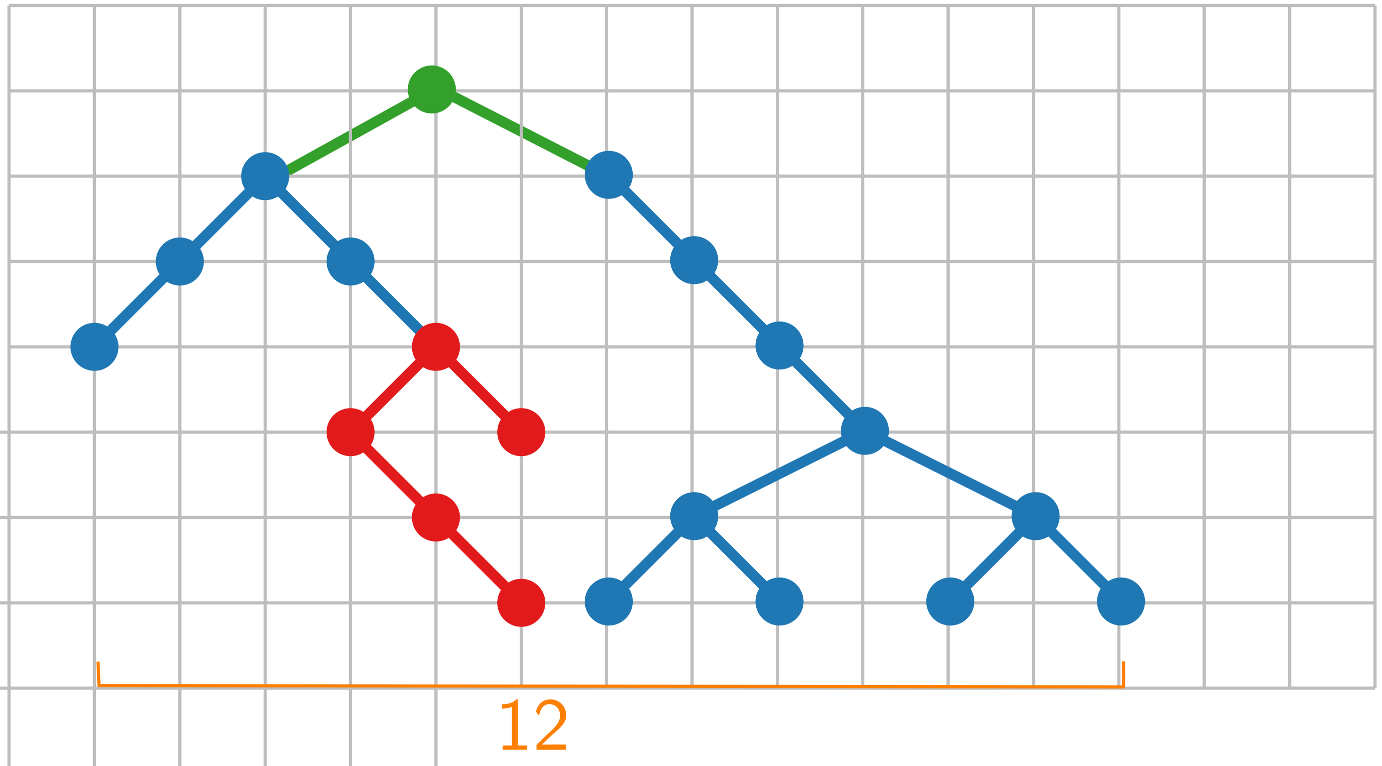


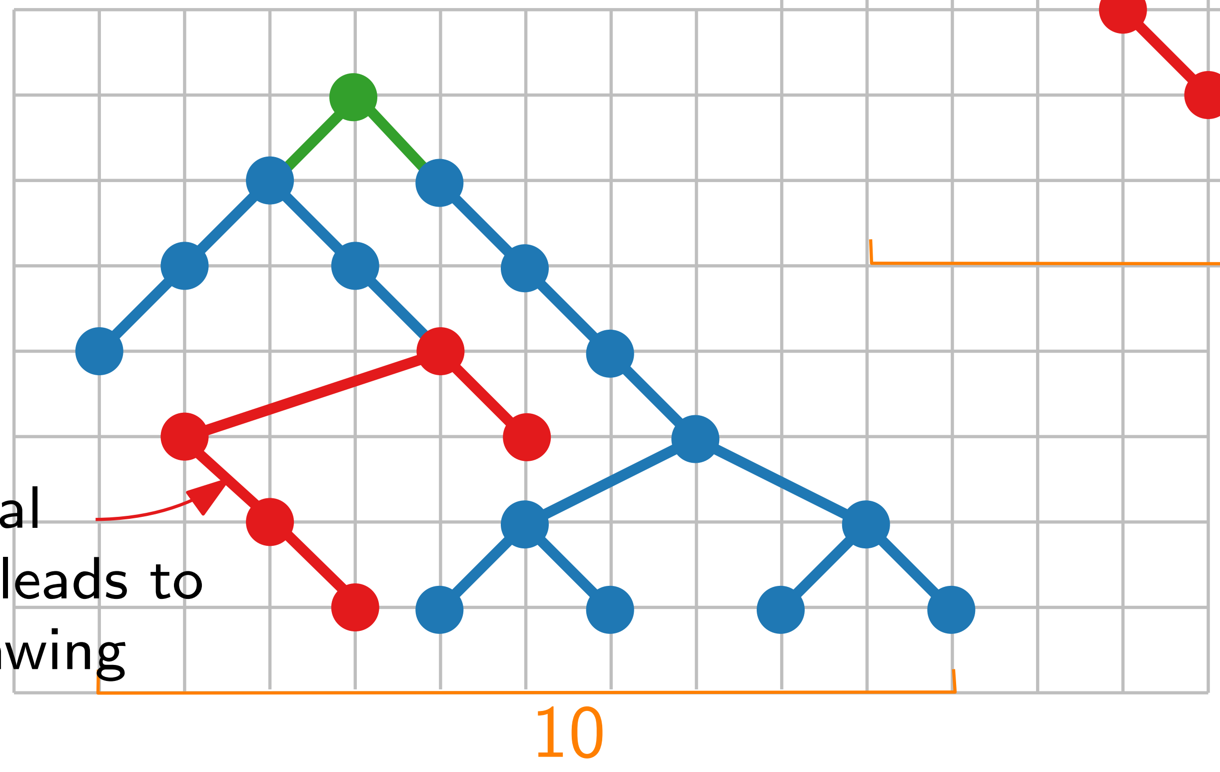


- Drawing with min width (but without the grid) can be constructed by an LP

Level-based layout – area

- Presented algorithm tries to minimise width
- Does not always achieve that!
- Divide-and-conquer strategy cannot achieve optimal width



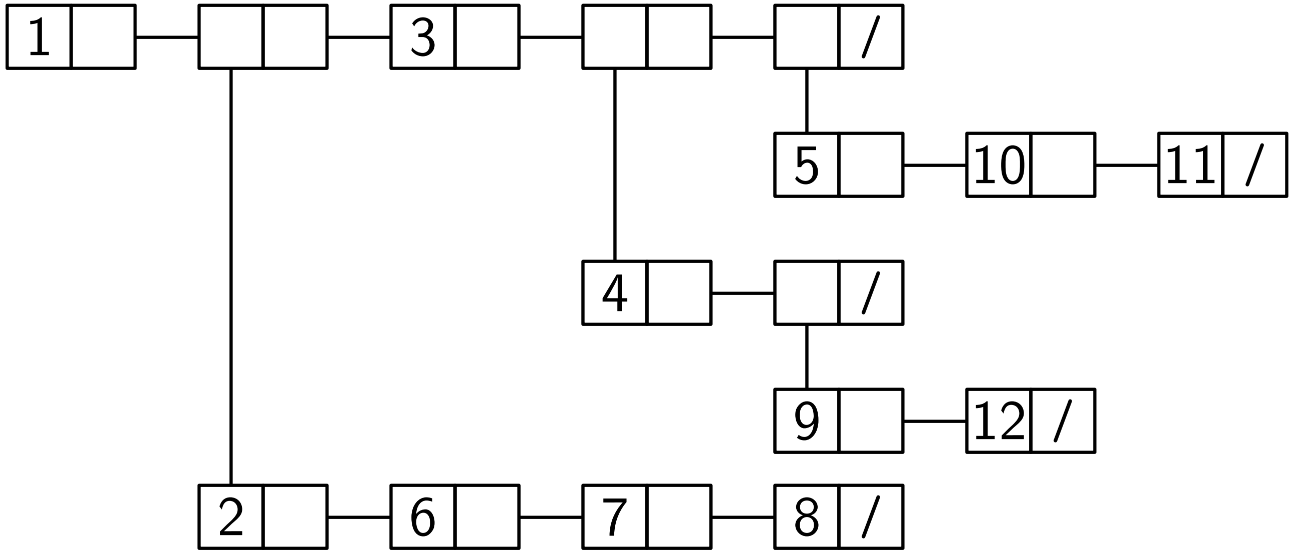


- Drawing with min width (but without the grid) can be constructed by an LP
- Problem is NP-hard on grid

Drawing-style: hv-drawings

Applications

- Cons cell diagram in LISP
- *Cons*(constructs) are memory objects which hold two values or pointers to values

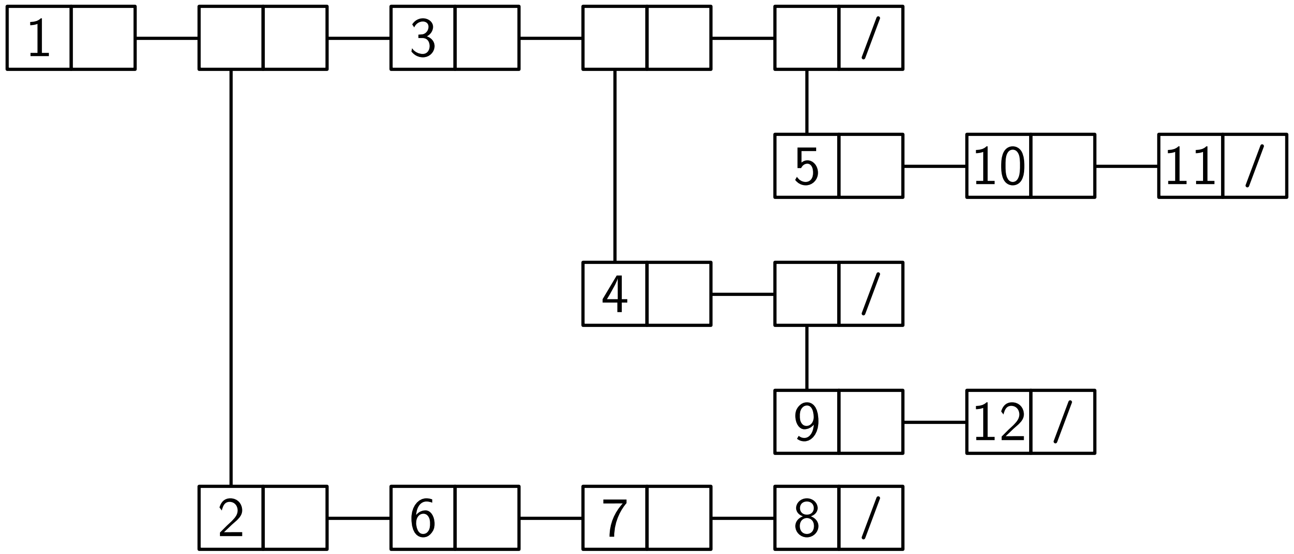


Source: after `gajon.org/trees-linked-lists-common-lisp/`

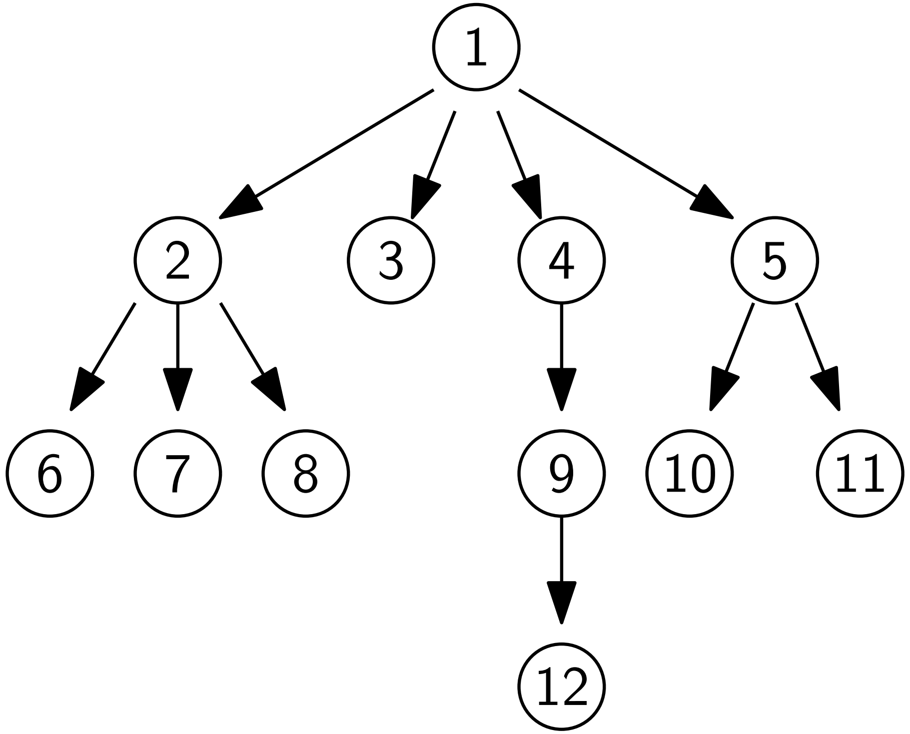
Drawing-style: hv-drawings

Applications

- Cons cell diagram in LISP
- *Cons*(constructs) are memory objects which hold two values or pointers to values



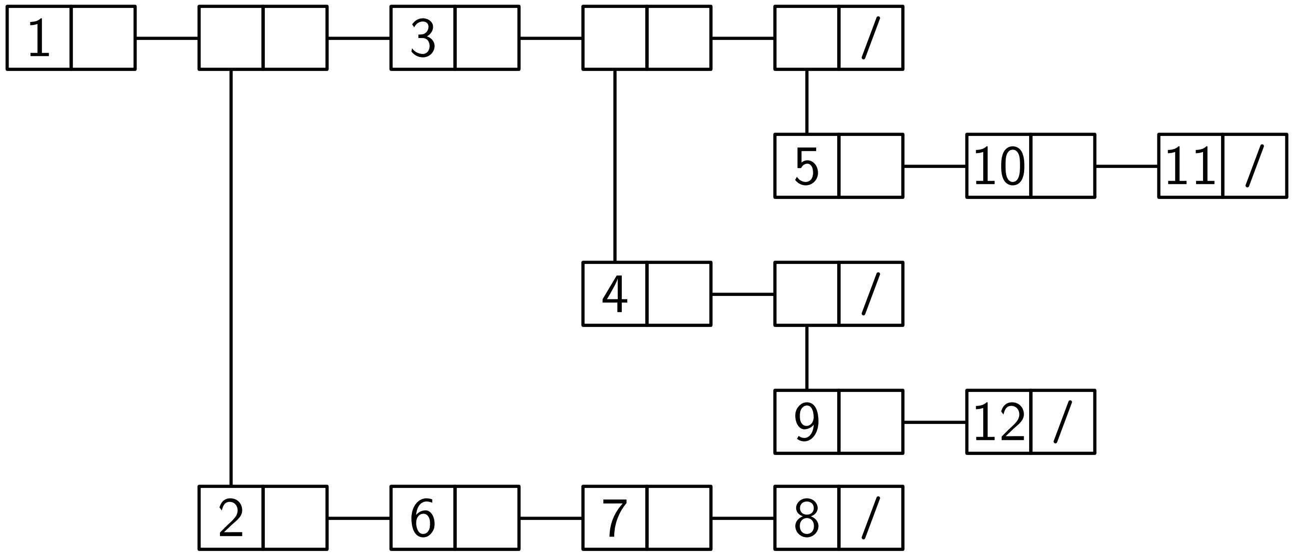
Source: after `gajon.org/trees-linked-lists-common-lisp/`



Drawing-style: hv-drawings

Applications

- Cons cell diagram in LISP
- *Cons*(constructs) are memory objects which hold two values or pointers to values



Source: after `gajon.org/trees-linked-lists-common-lisp/`

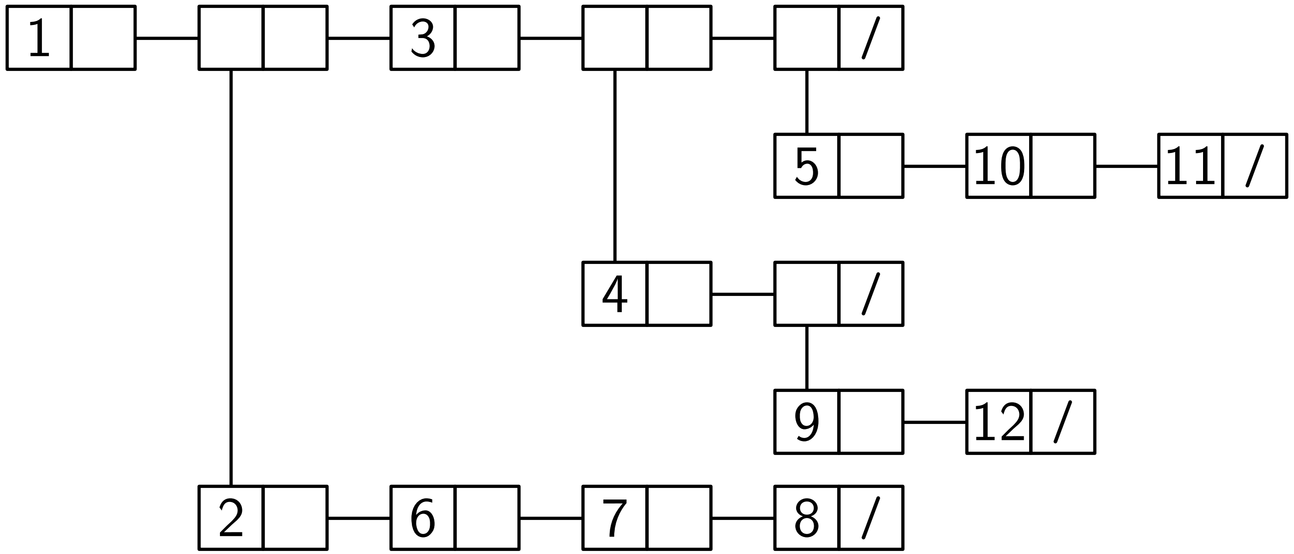
Drawing conventions

Drawing aesthetics

Drawing-style: hv-drawings

Applications

- Cons cell diagram in LISP
- *Cons*(constructs) are memory objects which hold two values or pointers to values



Source: after `gajon.org/trees-linked-lists-common-lisp/`

Drawing conventions

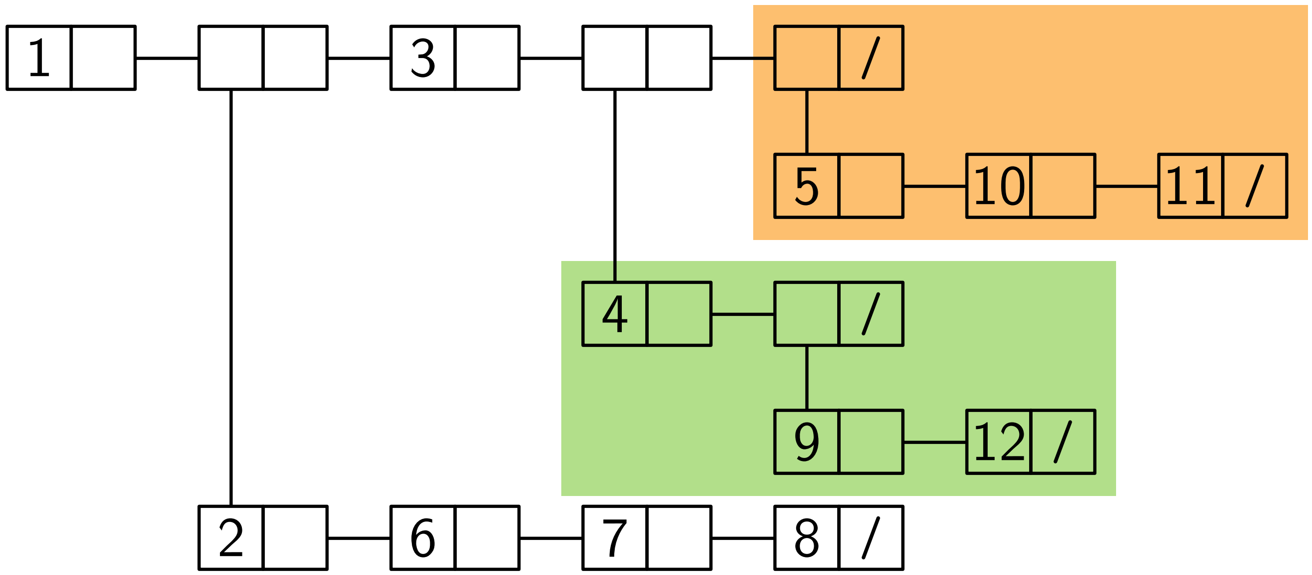
- Children are vertically and horizontally aligned with their parent

Drawing aesthetics

Drawing-style: hv-drawings

Applications

- Cons cell diagram in LISP
- *Cons*(constructs) are memory objects which hold two values or pointers to values



Source: after gajon.org/trees-linked-lists-common-lisp/

Drawing conventions

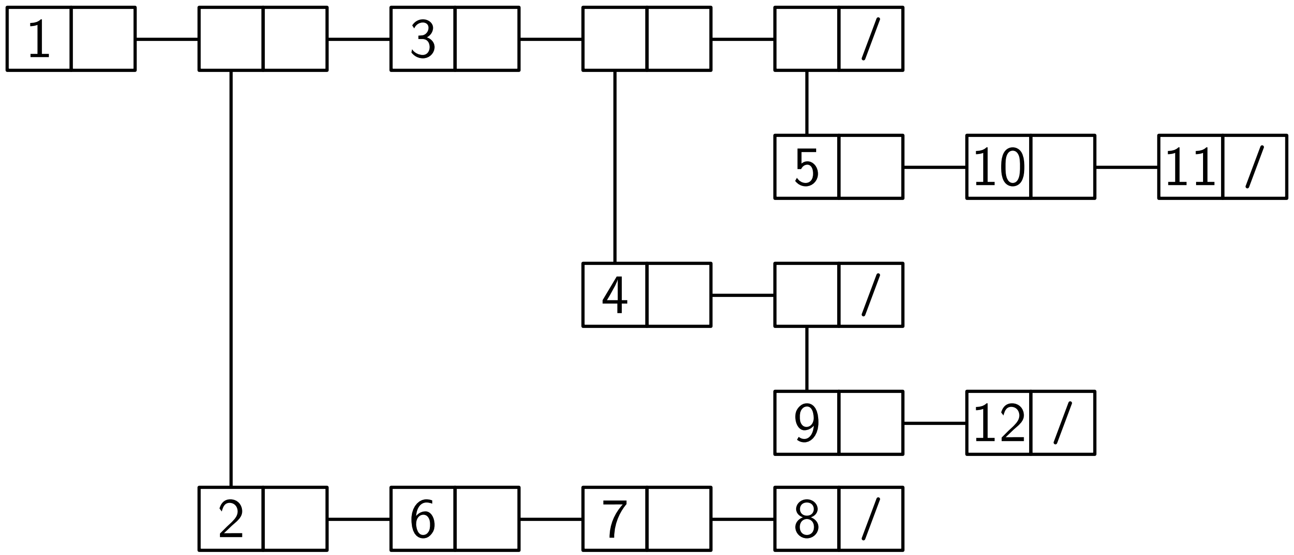
- Children are vertically and horizontally aligned with their parent
- The bounding boxes of the subtrees of the children are disjoint

Drawing aesthetics

Drawing-style: hv-drawings

Applications

- Cons cell diagram in LISP
- *Cons*(constructs) are memory objects which hold two values or pointers to values



Source: after gajon.org/trees-linked-lists-common-lisp/

Drawing conventions

- Children are vertically and horizontally aligned with their parent
- The bounding boxes of the subtrees of the children are disjoint

Drawing aesthetics

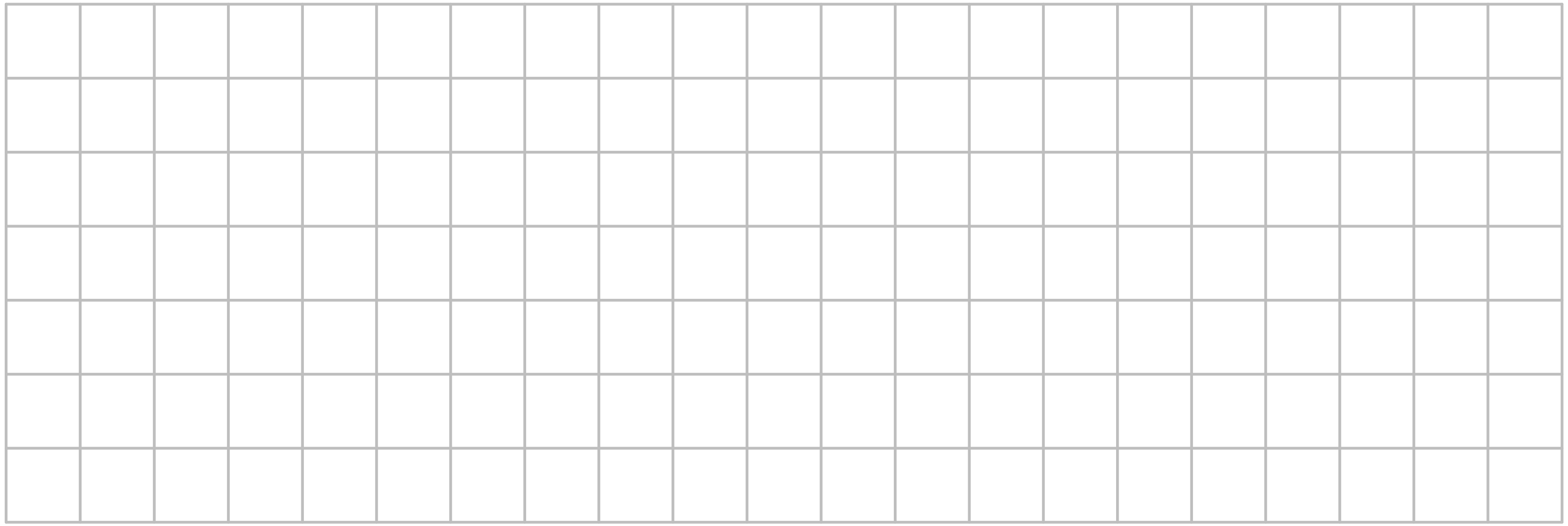
- Height, width, area

hv-drawings – algorithm

Input: A binary tree T
Output: A hv-drawing of T

Base case: 
Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:

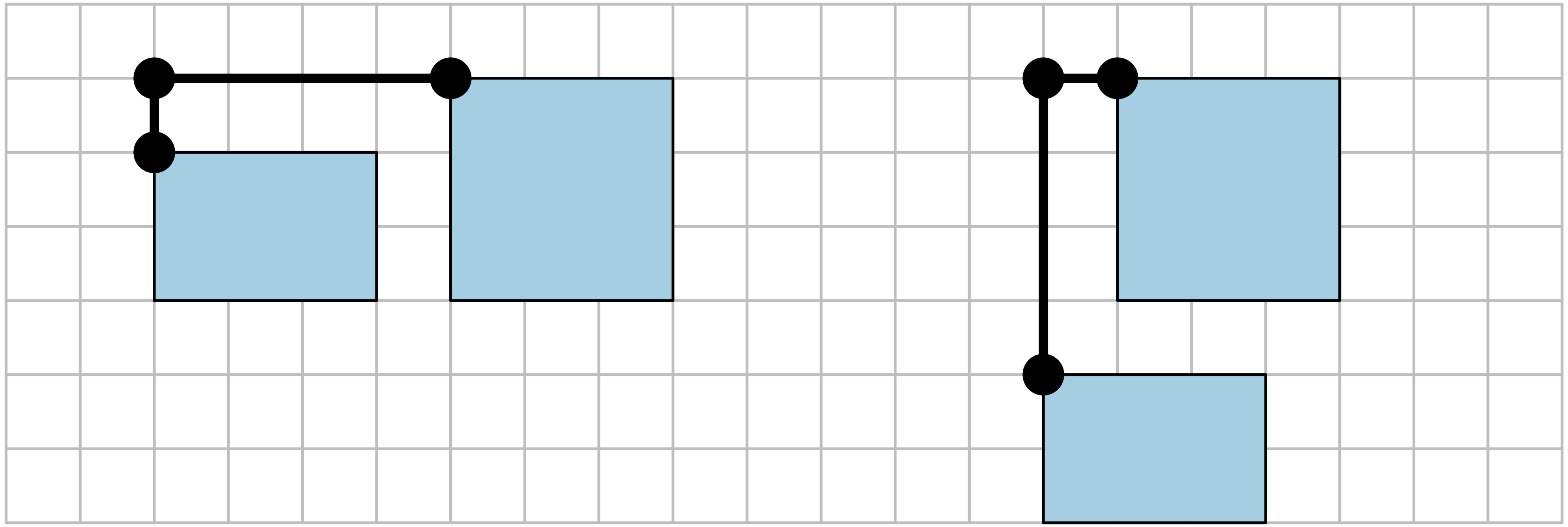


hv-drawings – algorithm

Input: A binary tree T
Output: A hv-drawing of T

Base case: ●
Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer: horizontal combination vertical combination



hv-drawing – right-heavy hv-layout

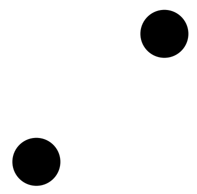
Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree $:=$ number of vertices

hv-drawing – right-heavy hv-layout

Right-heavy approach

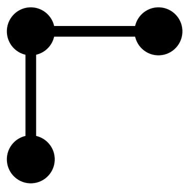
- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices



hv-drawing – right-heavy hv-layout

Right-heavy approach

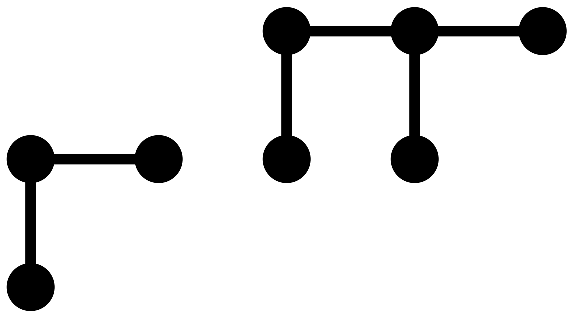
- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices



hv-drawing – right-heavy hv-layout

Right-heavy approach

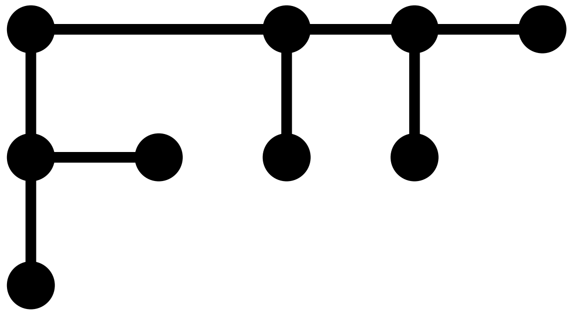
- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices



hv-drawing – right-heavy hv-layout

Right-heavy approach

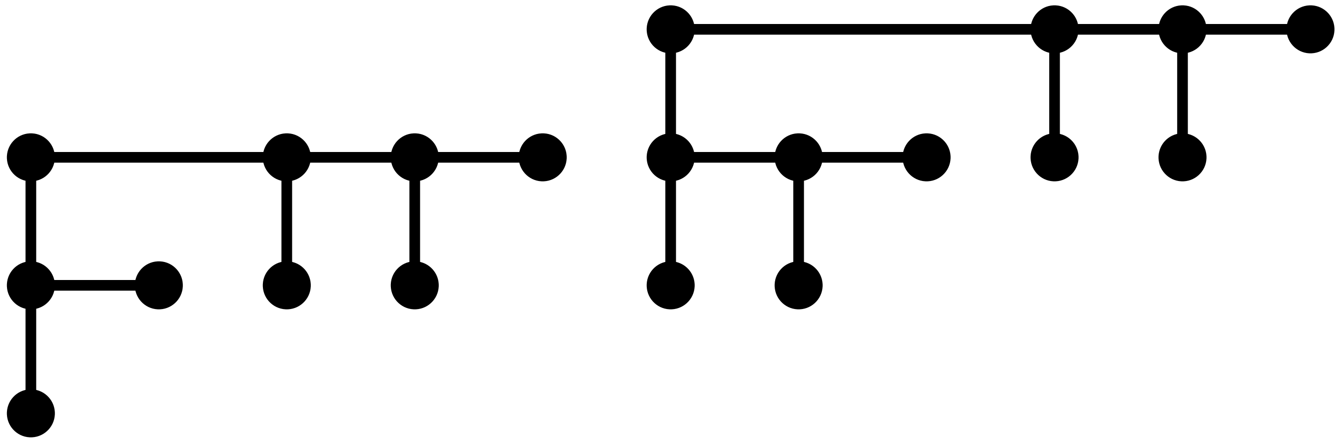
- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices



hv-drawing – right-heavy hv-layout

Right-heavy approach

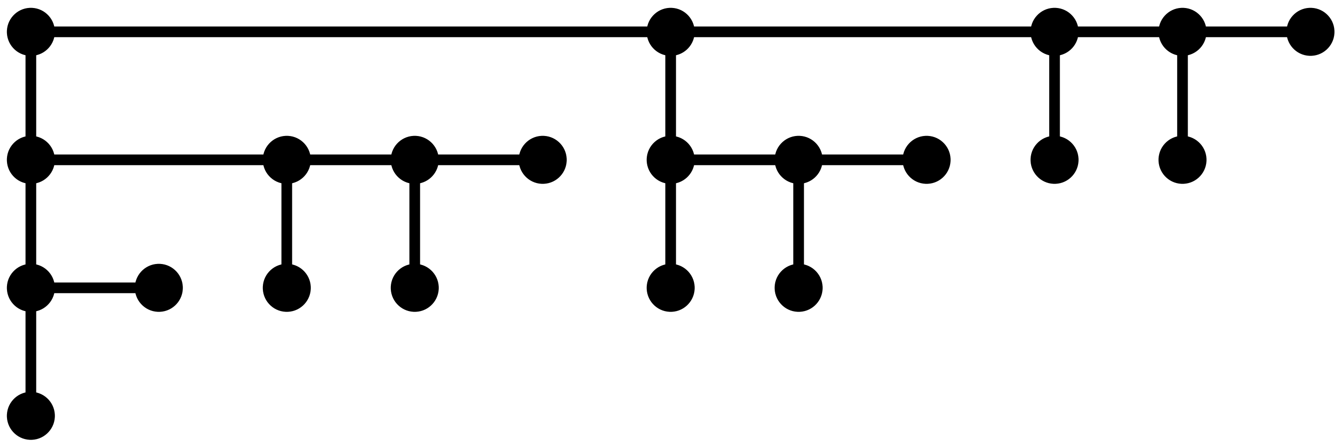
- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices



hv-drawing – right-heavy hv-layout

Right-heavy approach

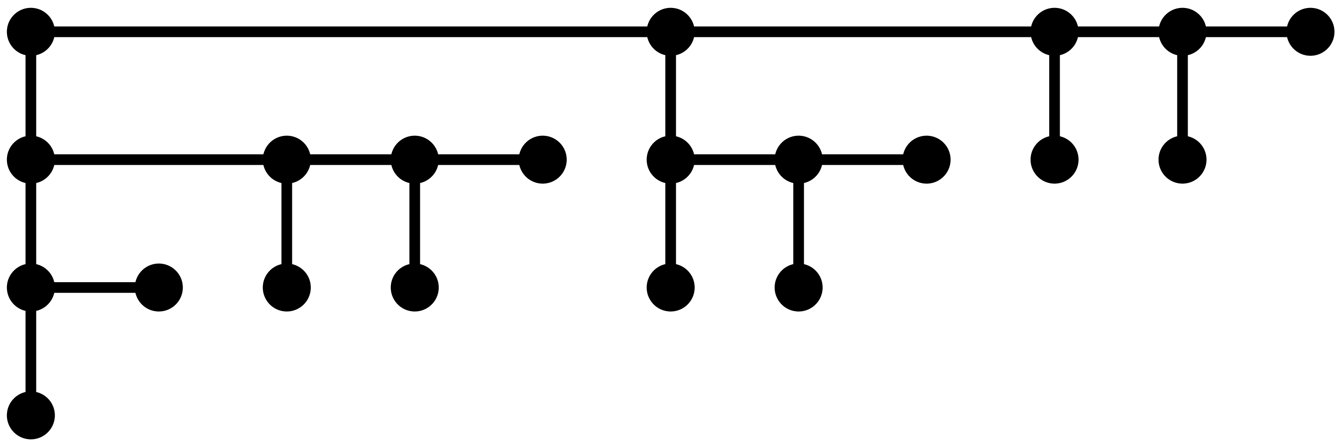
- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices



hv-drawing – right-heavy hv-layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices



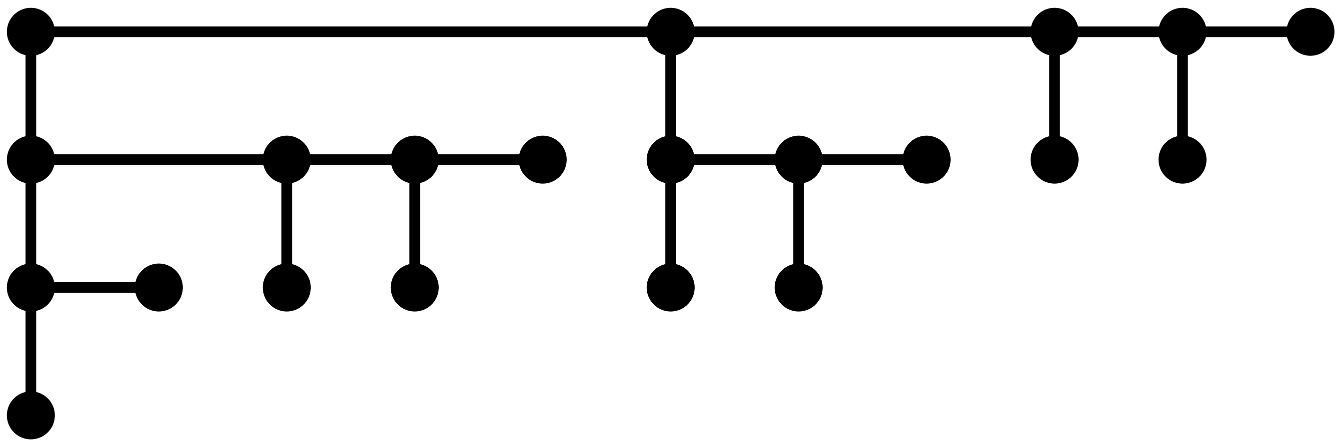
Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

- width at most and
- height at most

hv-drawing – right-heavy hv-layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices

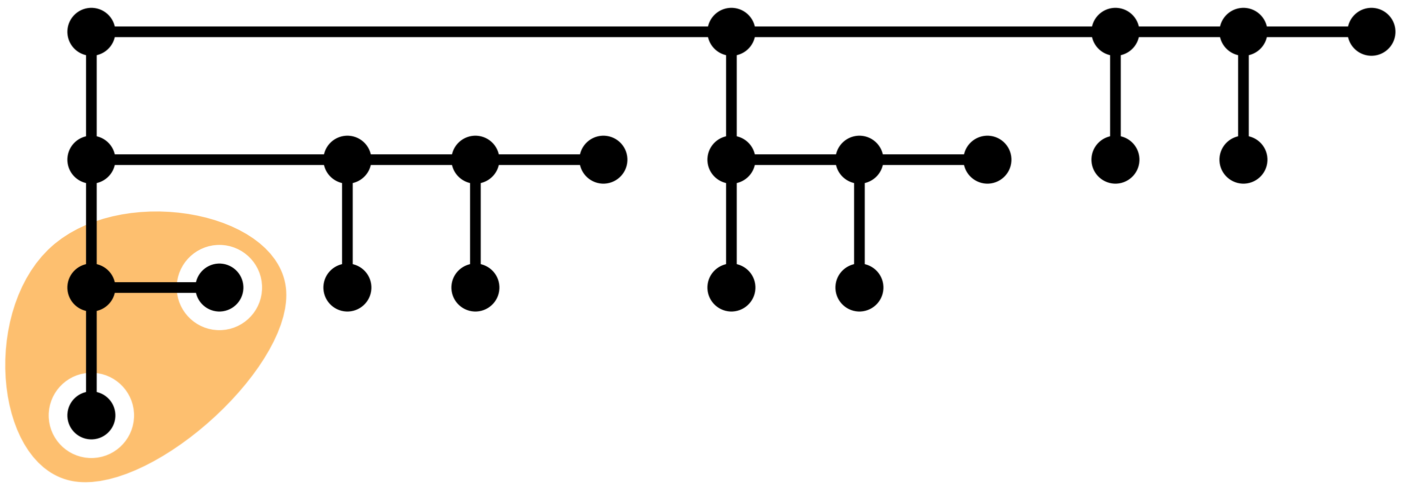


Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has
- width at most $n - 1$ and
- height at most

hv-drawing – right-heavy hv-layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices



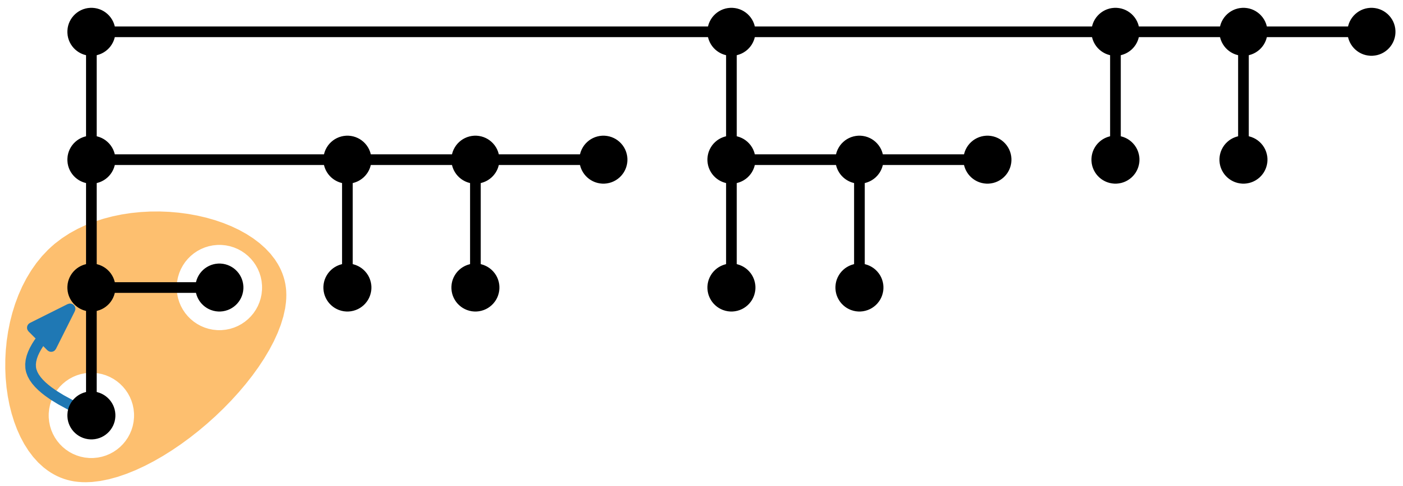
Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

- width at most $n - 1$ and
- height at most

hv-drawing – right-heavy hv-layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices



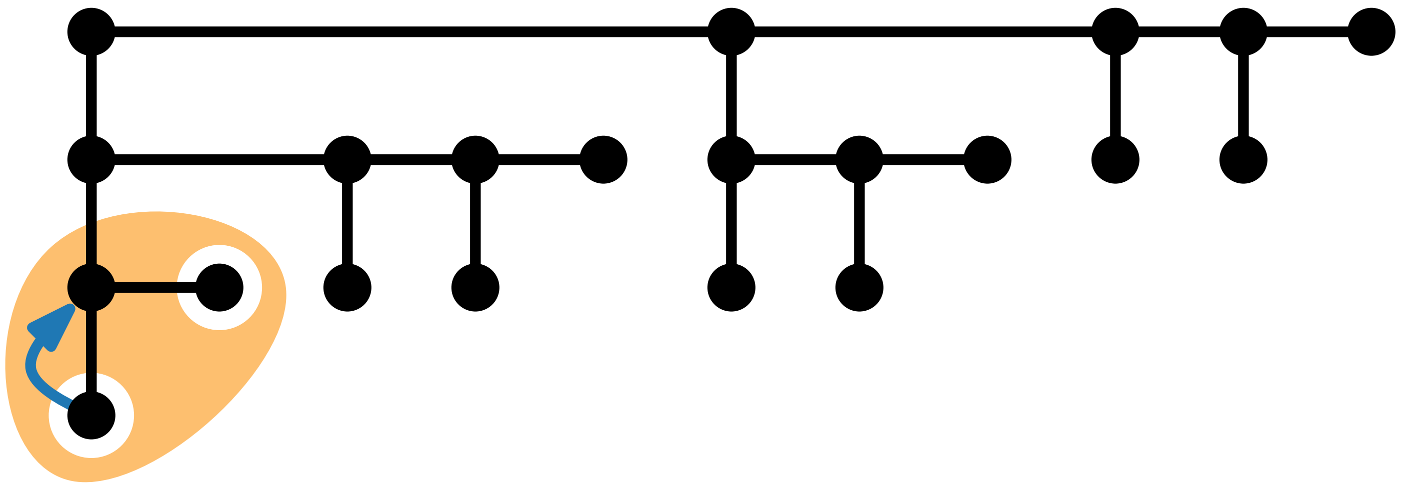
Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has
- width at most $n-1$ and
- height at most

hv-drawing – right-heavy hv-layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices

at least $\cdot 2$



Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

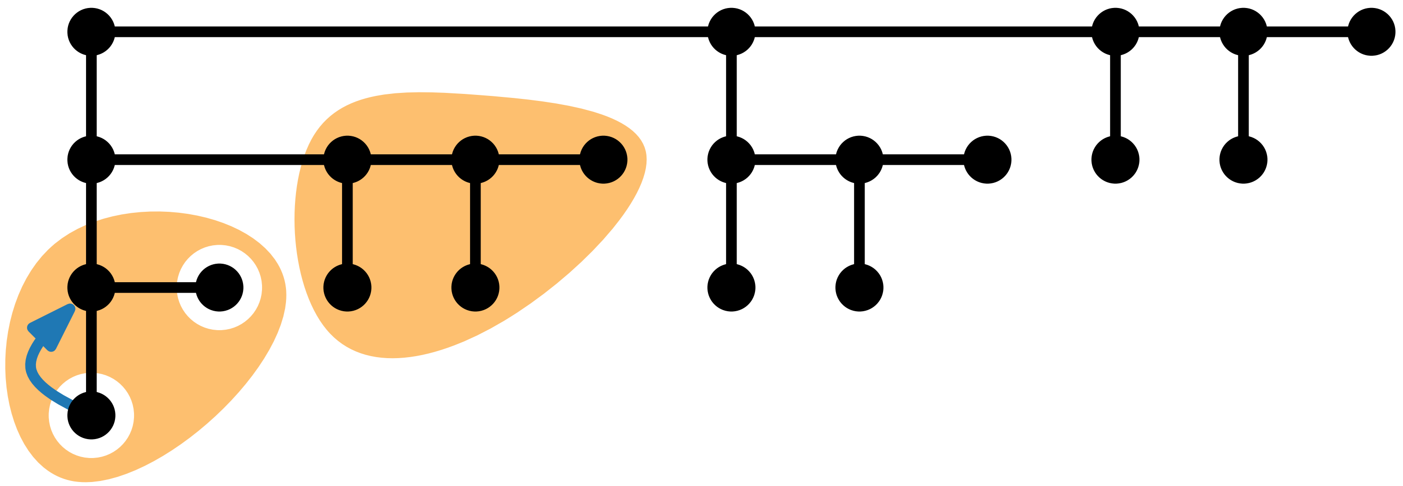
- width at most $n-1$ and
- height at most

hv-drawing – right-heavy hv-layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices

at least ·2

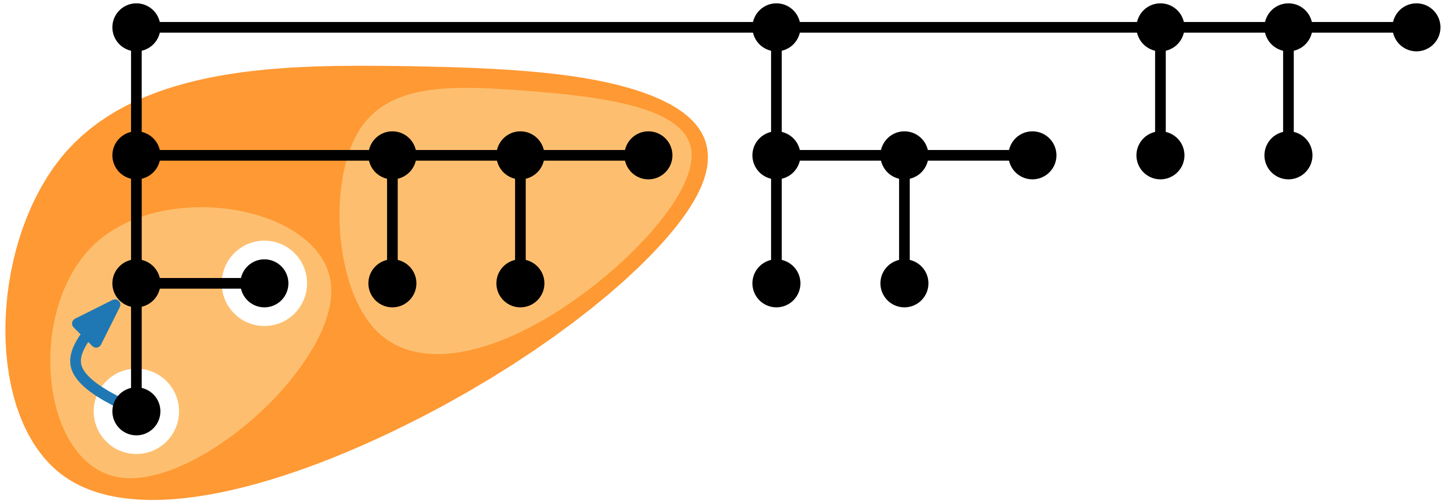


Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has
- width at most $n-1$ and
- height at most

hv-drawing – right-heavy hv-layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices



at least $\cdot 2$

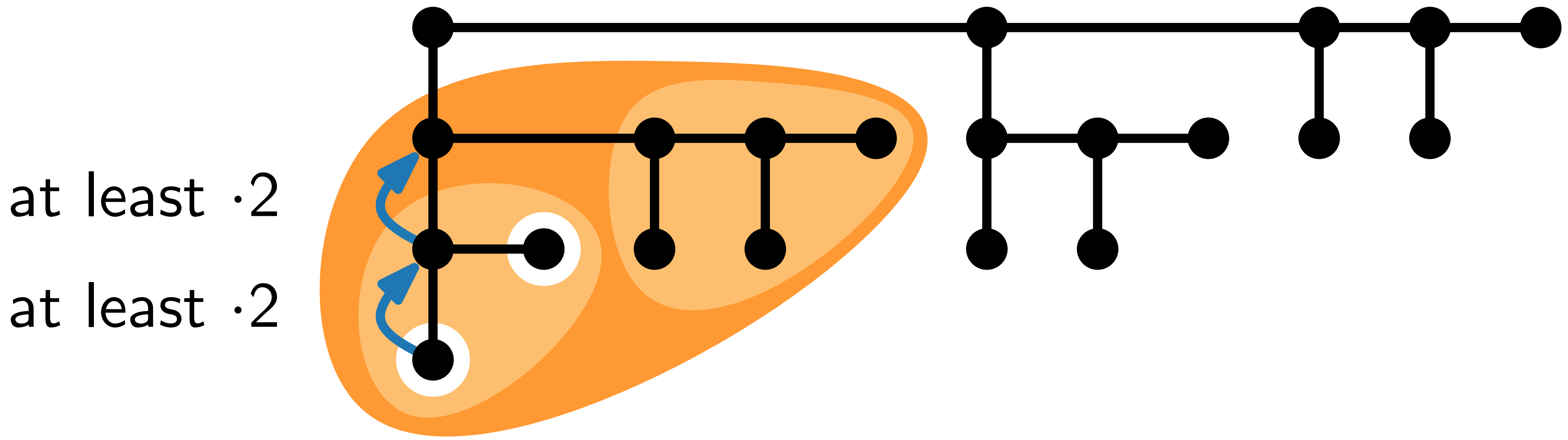
Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

- width at most $n-1$ and
- height at most

hv-drawing – right-heavy hv-layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices



Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has
- width at most $n-1$ and
- height at most

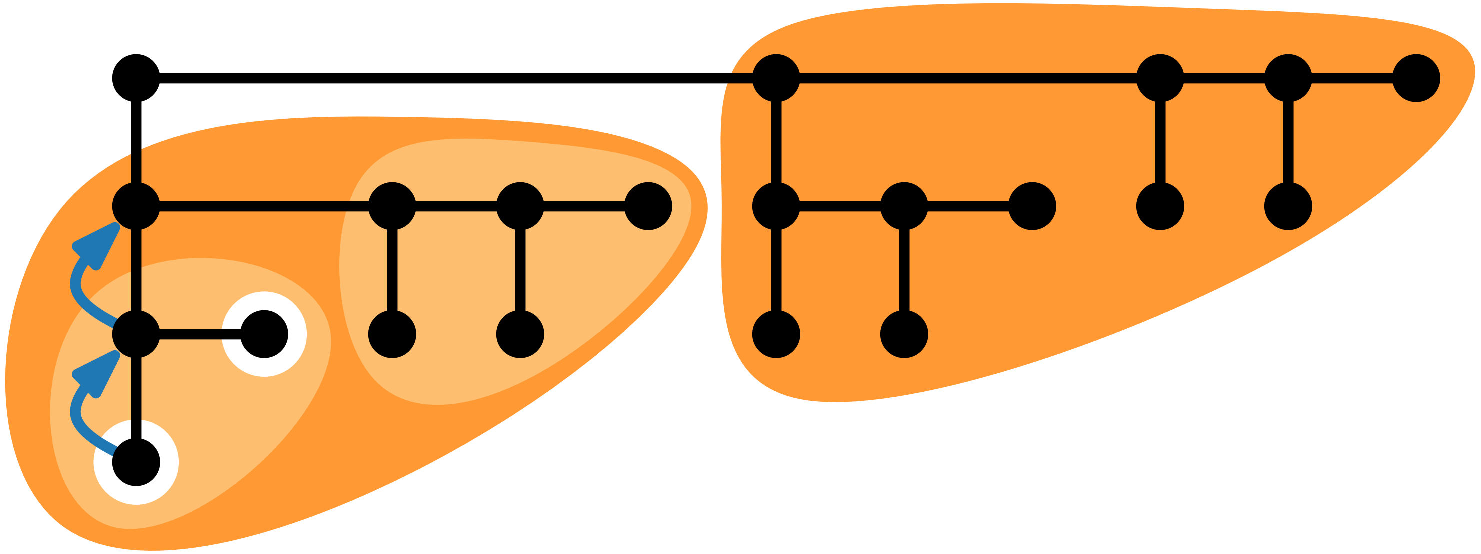
hv-drawing – right-heavy hv-layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices

at least $\cdot 2$

at least $\cdot 2$



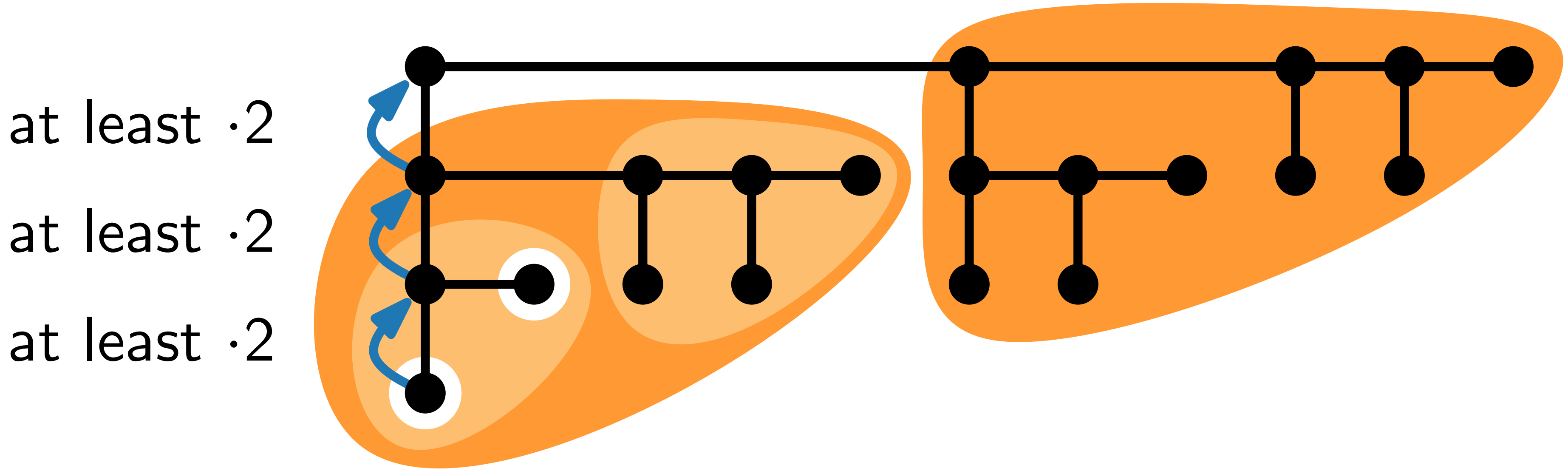
Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

- width at most $n - 1$ and
- height at most

hv-drawing – right-heavy hv-layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices



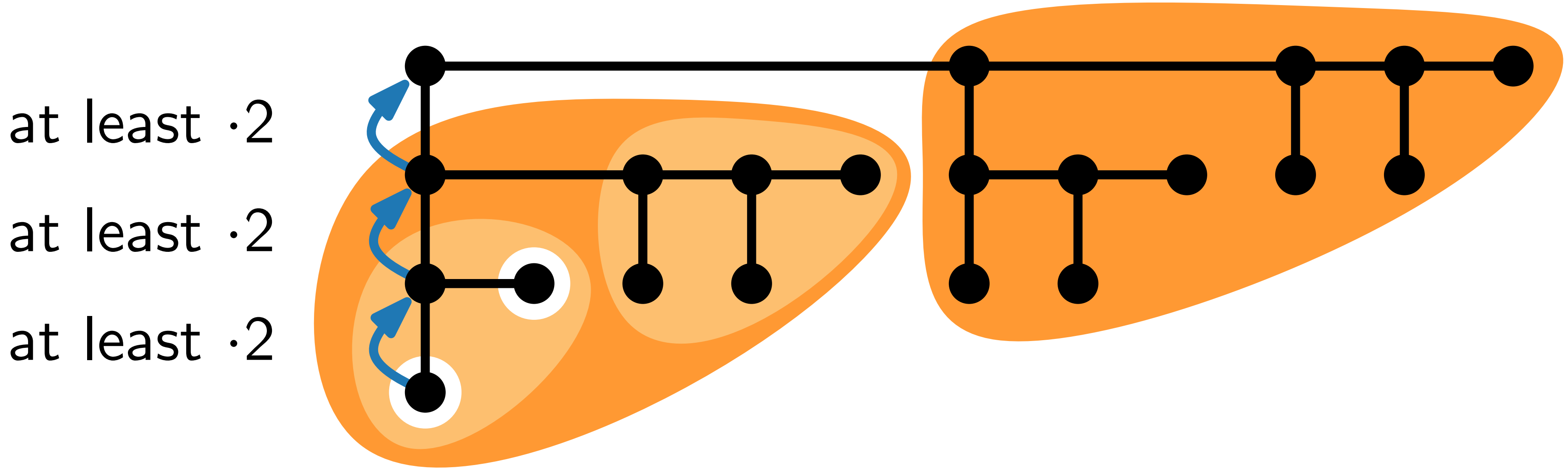
Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

- width at most $n - 1$ and
- height at most

hv-drawing – right-heavy hv-layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices



Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

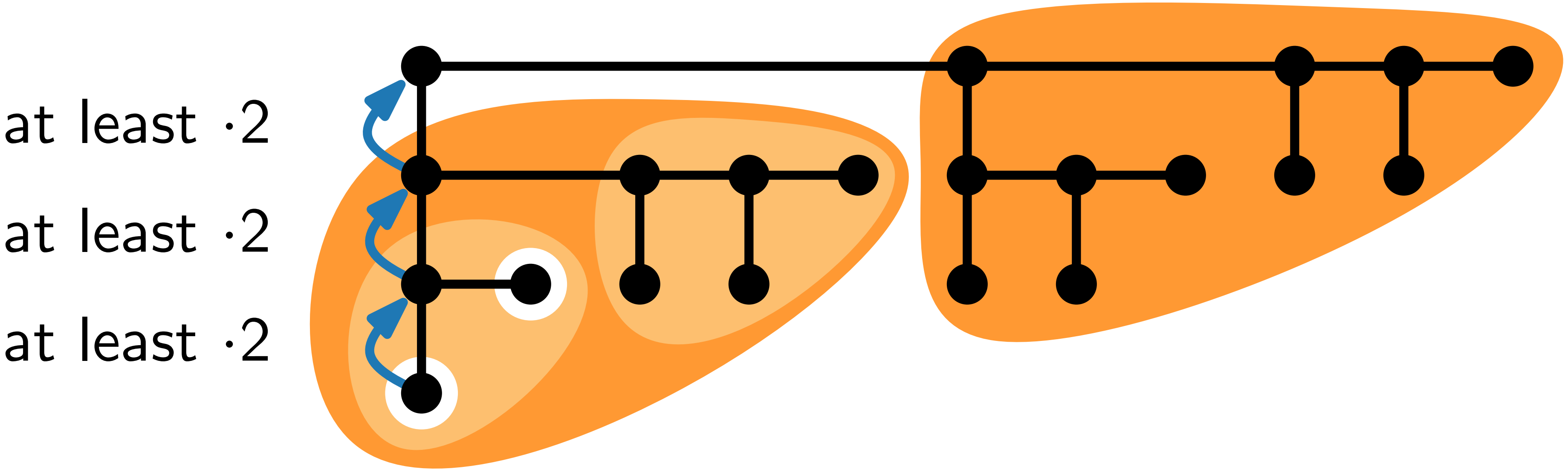
- width at most $n - 1$ and
- height at most $\log n$.

hv-drawing – right-heavy hv-layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right
  - Size of subtree := number of vertices

How to implement this in linear time?



Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

- width at most $n - 1$ and
- height at most $\log n$.

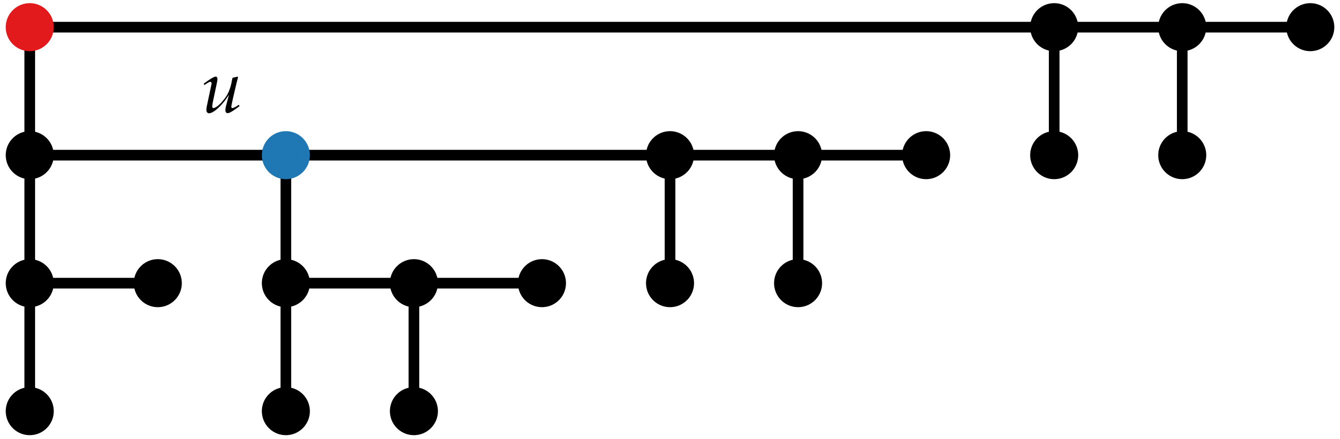
Computing right-heavy hv-layout in linear time

- At each node u we store the 5-tuple:

$$u : (x_u, y_u, W_u, H_u, s_u)$$

  where:

  - x_u, y_u are the x and y coordinates of u



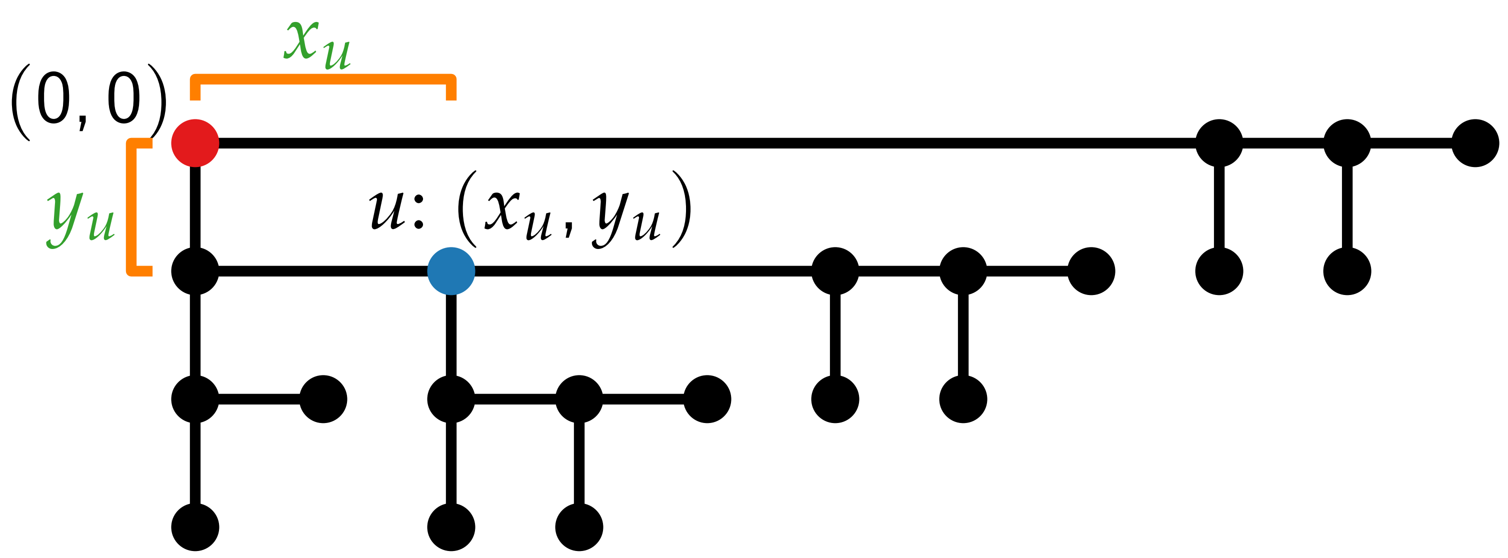
Computing right-heavy hv-layout in linear time

- At each node u we store the 5-tuple:

$$u : (x_u, y_u, W_u, H_u, s_u)$$

  where:

  - x_u, y_u are the x and y coordinates of u



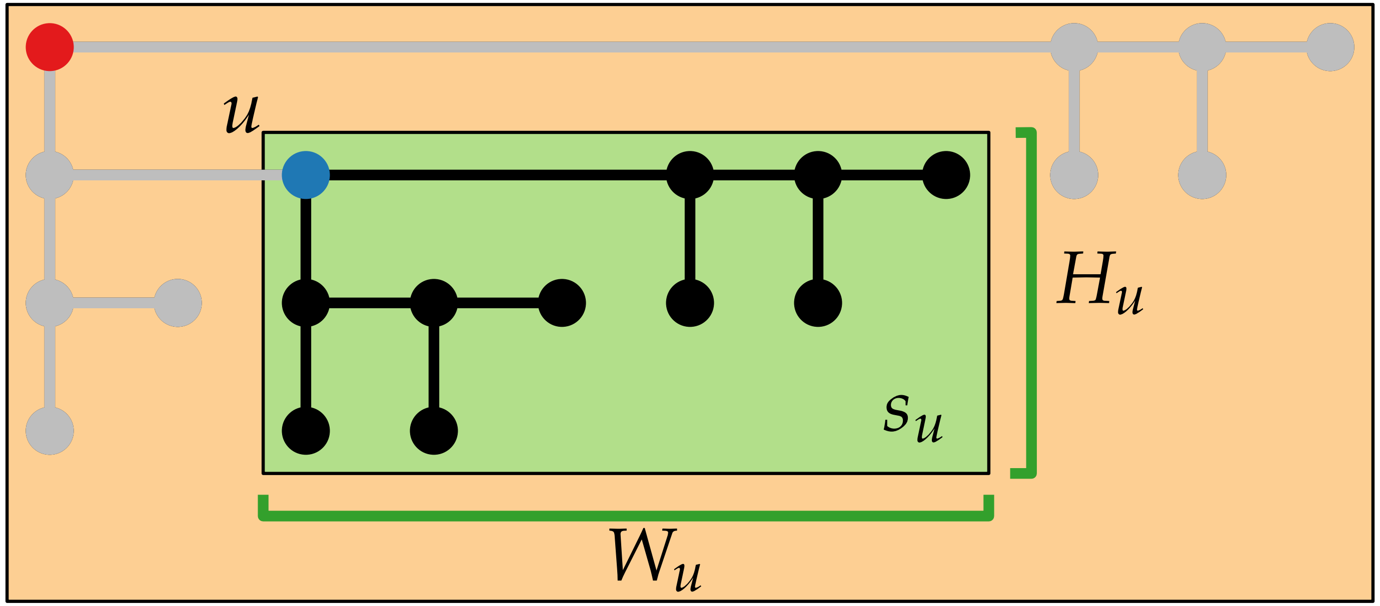
Computing right-heavy hv-layout in linear time

- At each node u we store the 5-tuple:

$$u : (x_u, y_u, W_u, H_u, s_u)$$

  where:

  - x_u, y_u are the x and y coordinates of u
  - W_u is the width of the layout of subtree T_u
  - H_u is the height of the layout of subtree T_u
  - s_u is the size of T_u



Computing right-heavy hv-layout in linear time

- Compute in a bottom-up fashion (by a post-order traversal) s_u, W_u and H_u

Computing right-heavy hv-layout in linear time

- Compute in a bottom-up fashion (by a post-order traversal) s_u, W_u and H_u

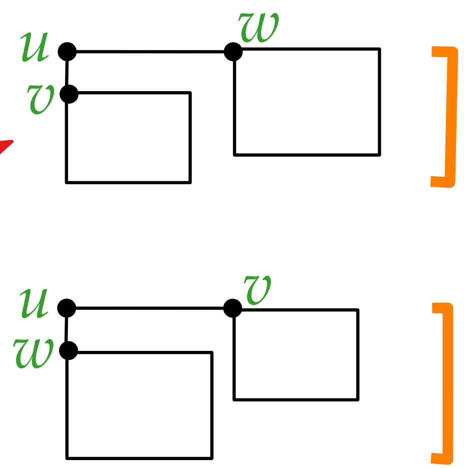
$u :$
- $s_u = s_v + s_w + 1$

Computing right-heavy hv-layout in linear time

- Compute in a bottom-up fashion (by a post-order traversal) s_u, W_u and H_u

u :

- $s_u = s_v + s_w + 1$
- **if** $(s_v < s_w)$
  $H_u = \max(H_v + 1, H_w)$
  else
  $H_u = \max(H_w + 1, H_v)$

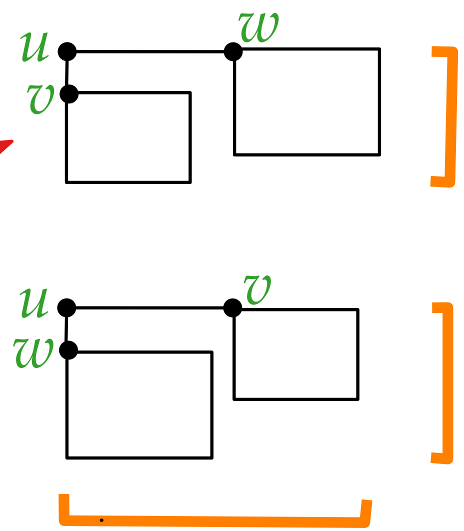


Computing right-heavy hv-layout in linear time

- Compute in a bottom-up fashion (by a post-order traversal) s_u, W_u and H_u

u :

- $s_u = s_v + s_w + 1$
- **if** $(s_v < s_w)$
  $H_u = \max(H_v + 1, H_w)$
  else
  $H_u = \max(H_w + 1, H_v)$
- $W_u = W_v + W_w + 1$



Computing right-heavy hv-layout in linear time

- Compute in a top-down fashion (by a pre-order traversal) x_u and y_u

Computing right-heavy hv-layout in linear time

- Compute in a top-down fashion (by a pre-order traversal) x_u and y_u

$r:$ • $x_r = 0, \quad y_r = 0$

Computing right-heavy hv-layout in linear time

- Compute in a top-down fashion (by a pre-order traversal) x_u and y_u

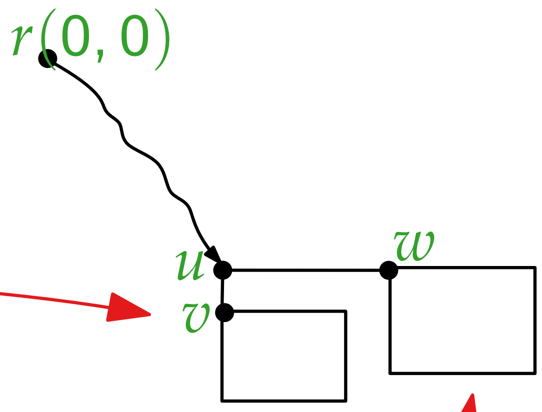
r :
- $x_r = 0, \quad y_r = 0$

u :
- For subtree rooted at v and placed below u:

$$x_v = x_u$$
$$y_v = y_u - 1$$

- For subtree rooted at w and placed to the right of u:

$$x_w = x_u + W_v + 1$$
$$y_w = y_u$$



Computing right-heavy hv-layout in linear time

- Compute in a top-down fashion (by a pre-order traversal) x_u and y_u

r :
- $x_r = 0, \quad y_r = 0$

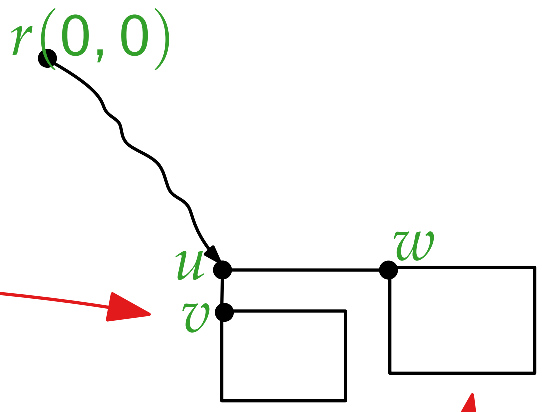
u :
- For subtree rooted at v and placed below u:

$$x_v = x_u$$
$$y_v = y_u - 1$$

- For subtree rooted at w and placed to the right of u:

$$x_w = x_u + W_v + 1$$
$$y_w = y_u$$

Total time: $O(n)$



hv-drawing – result (1)

Theorem.
Let T be a binary tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is hv-drawing (planar, orthogonal)
- Width is at most $n - 1$
- Height is at most $\log n$
- Area is in $\mathcal{O}(n \log n)$

hv-drawing – result (1)

Theorem.
Let T be a binary tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is hv-drawing (planar, orthogonal)
- Width is at most $n-1$
- Height is at most $\log n$
- Area is in $\mathcal{O}(n \log n)$
- Simply and axially isomorphic subtrees have congruent drawings up to translation

hv-drawing – result (1)

Theorem.
Let T be a binary tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is hv-drawing (planar, orthogonal)
- Width is at most $n - 1$
- Height is at most $\log n$
- Area is in $\mathcal{O}(n \log n)$
- Simply and axially isomorphic subtrees have congruent drawings up to translation

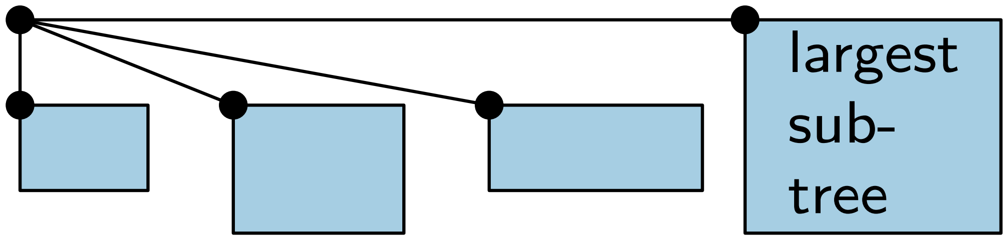
Bad aspect ratio $\Omega(n/\log n)$

hv-drawing – result (1)

Theorem.
Let T be a binary tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is hv-drawing (planar, orthogonal)
- Width is at most $n - 1$
- Height is at most $\log n$
- Area is in $\mathcal{O}(n \log n)$
- Simply and axially isomorphic subtrees have congruent drawings up to translation

General rooted tree



hv-drawing – balanced layout

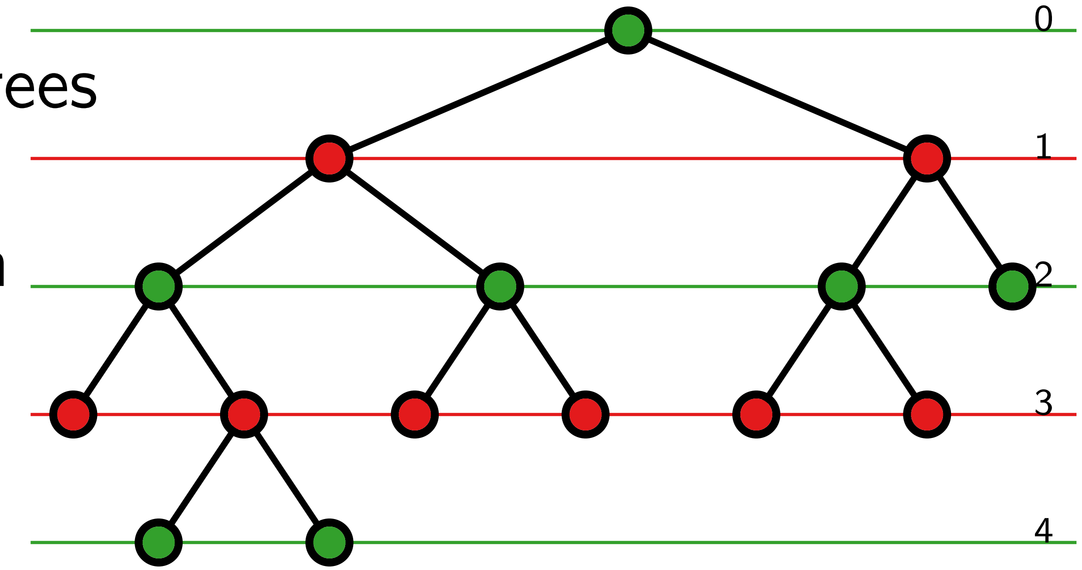
Balanced approach

- Recursively compute layout for left and right subtrees
- Apply
  - horizontal combination if vertex is at odd depth
  - vertical combination if vertex is at even depth

hv-drawing – balanced layout

Balanced approach

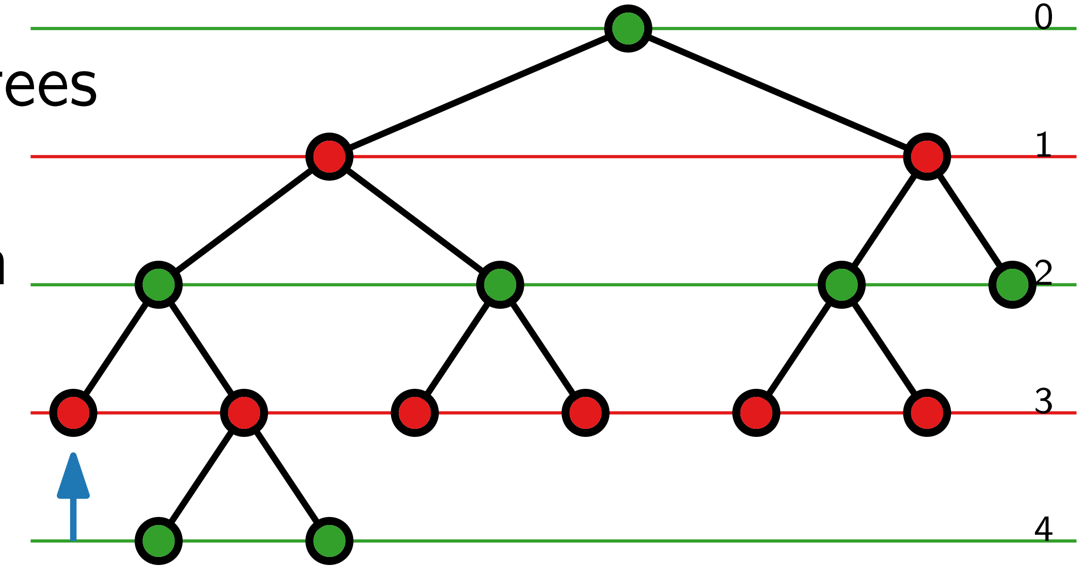
- Recursively compute layout for left and right subtrees
- Apply
  - horizontal combination if vertex is at odd depth
  - vertical combination if vertex is at even depth

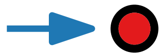


hv-drawing – balanced layout

Balanced approach

- Recursively compute layout for left and right subtrees
- Apply
  - horizontal combination if vertex is at odd depth
  - vertical combination if vertex is at even depth

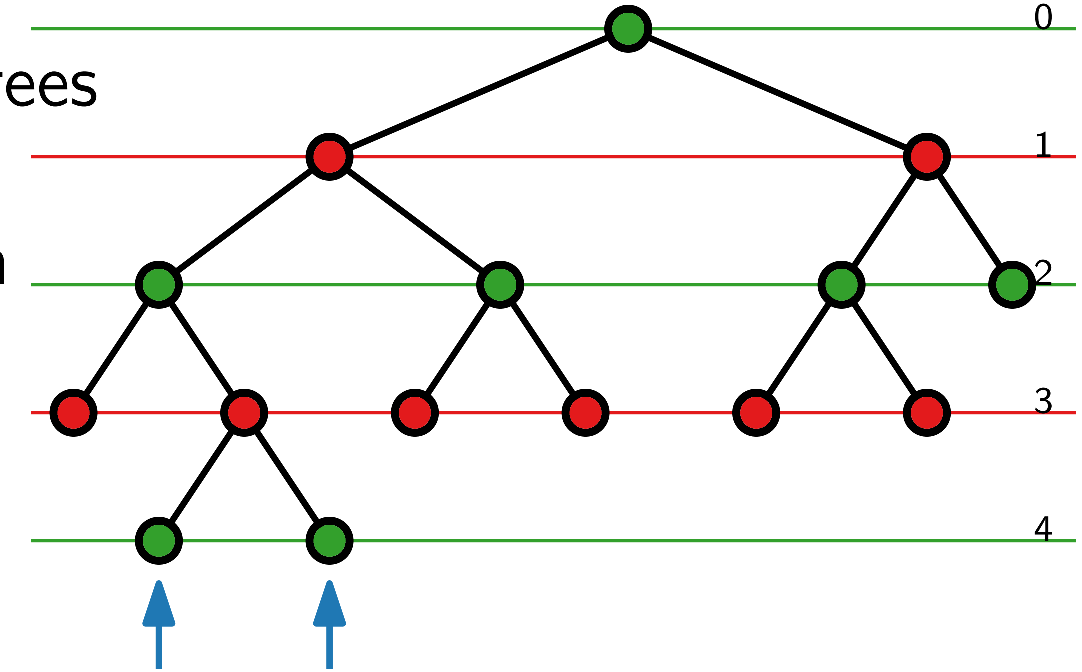


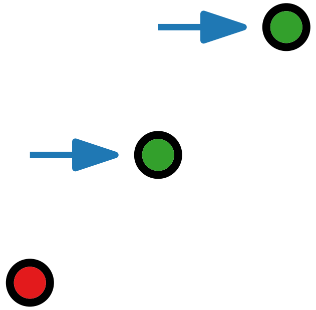


hv-drawing – balanced layout

Balanced approach

- Recursively compute layout for left and right subtrees
- Apply
  - horizontal combination if vertex is at odd depth
  - vertical combination if vertex is at even depth

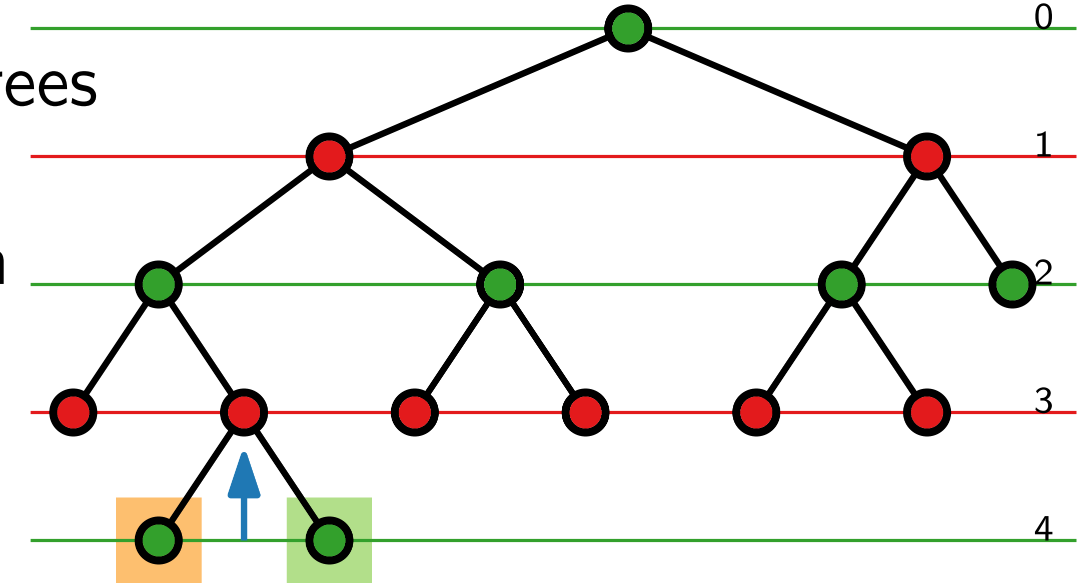


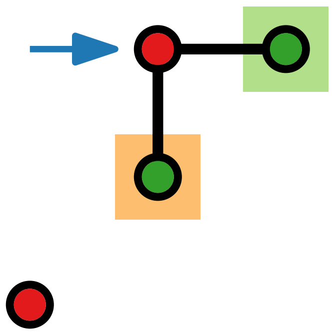


hv-drawing – balanced layout

Balanced approach

- Recursively compute layout for left and right subtrees
- Apply
  - horizontal combination if vertex is at odd depth
  - vertical combination if vertex is at even depth

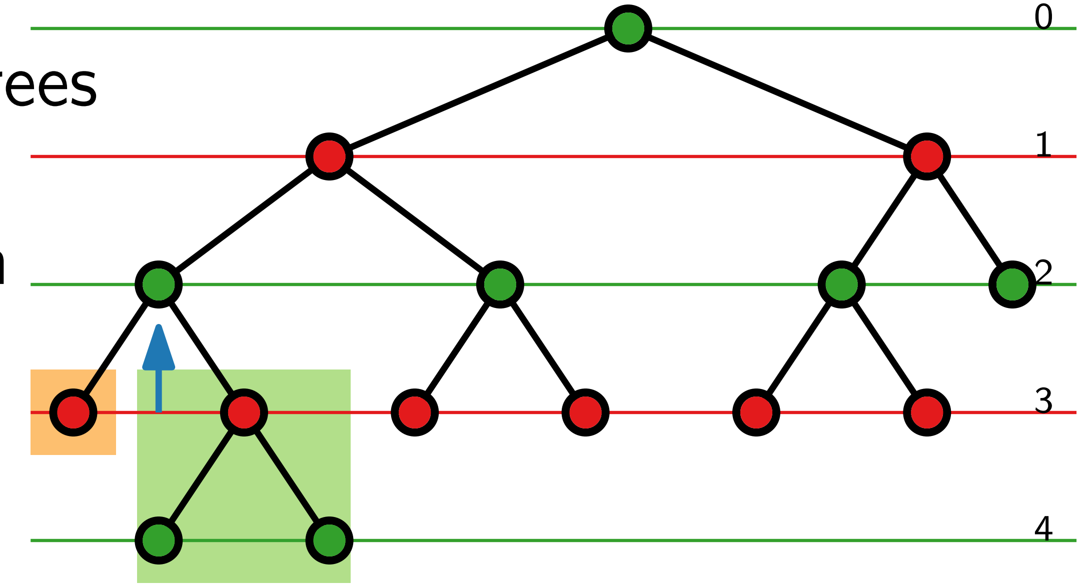


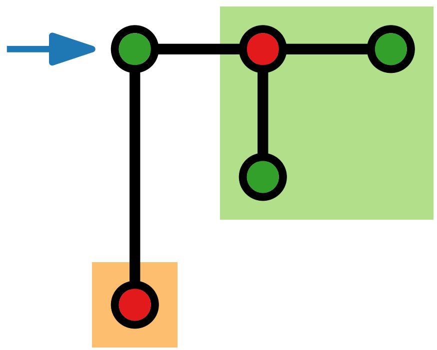


hv-drawing – balanced layout

Balanced approach

- Recursively compute layout for left and right subtrees
- Apply
  - horizontal combination if vertex is at odd depth
  - vertical combination if vertex is at even depth

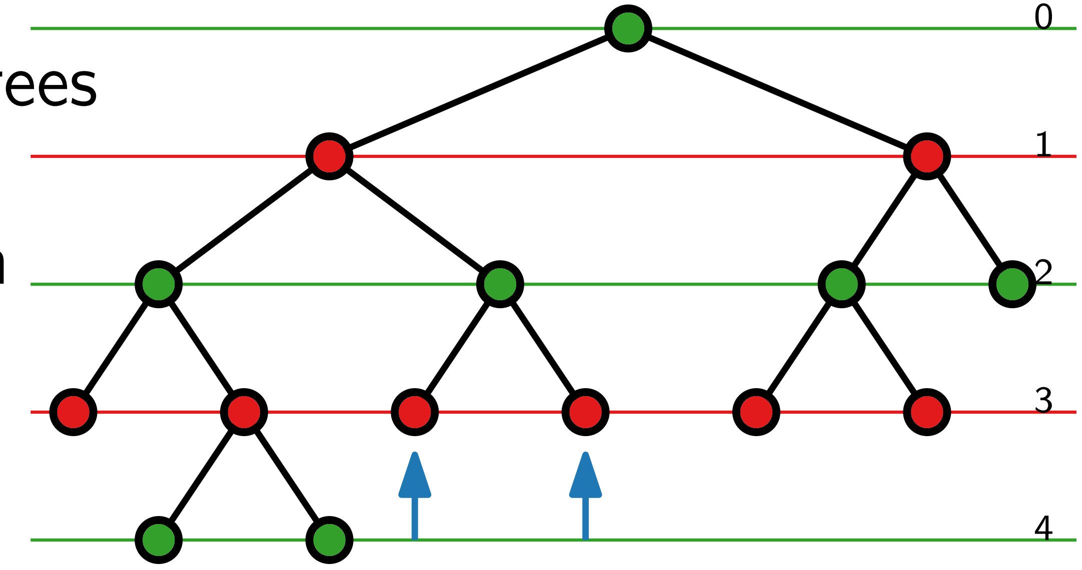


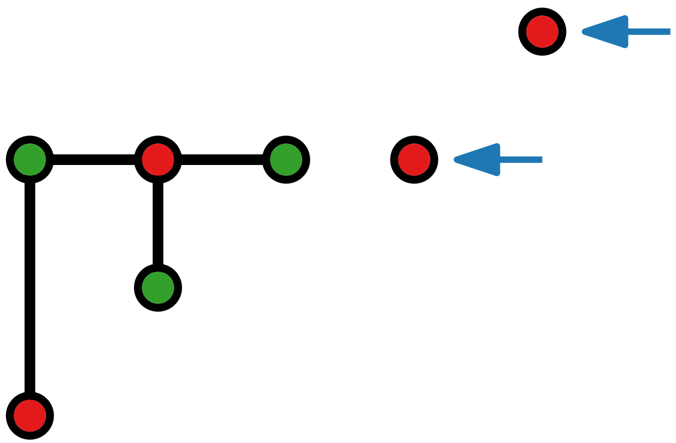


hv-drawing – balanced layout

Balanced approach

- Recursively compute layout for left and right subtrees
- Apply
  - horizontal combination if vertex is at odd depth
  - vertical combination if vertex is at even depth

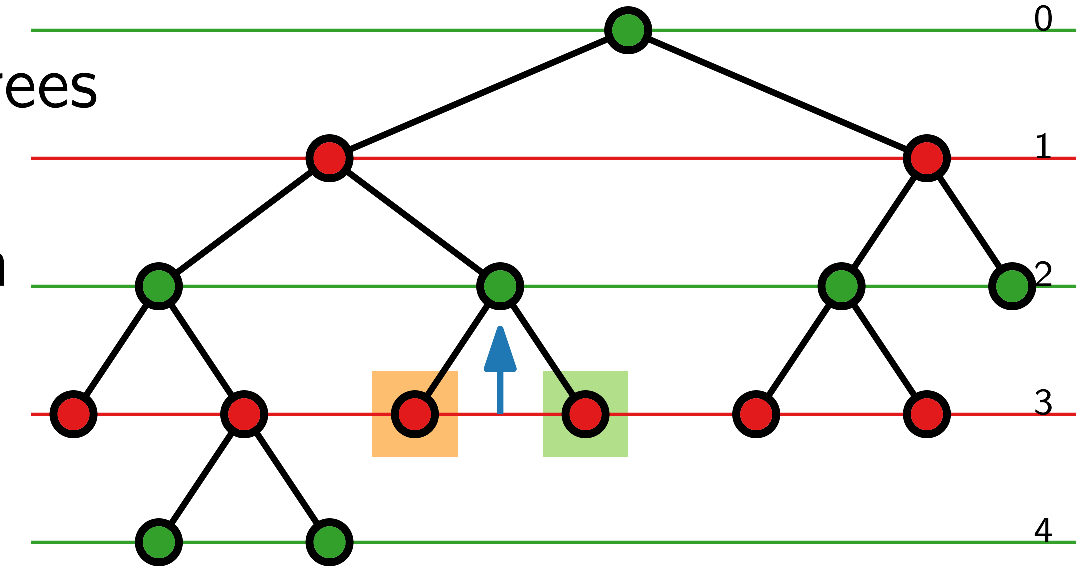


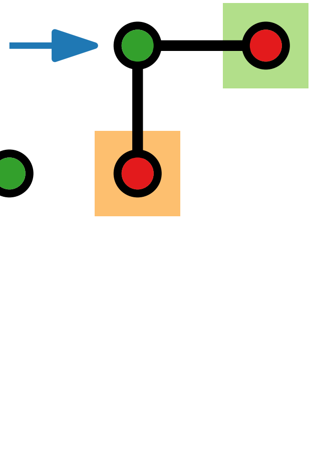


hv-drawing – balanced layout

Balanced approach

- Recursively compute layout for left and right subtrees
- Apply
  - horizontal combination if vertex is at odd depth
  - vertical combination if vertex is at even depth

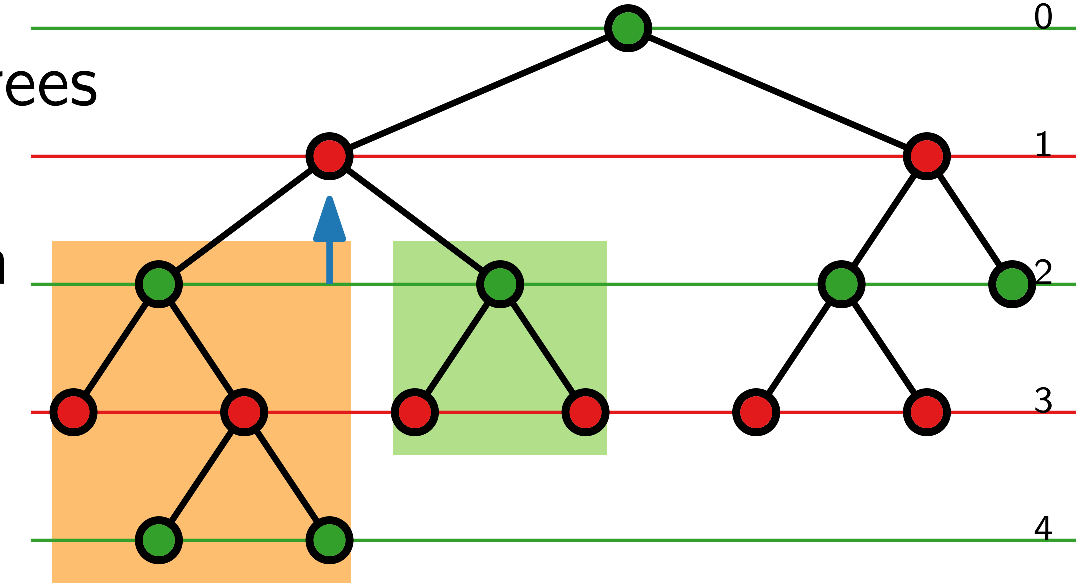


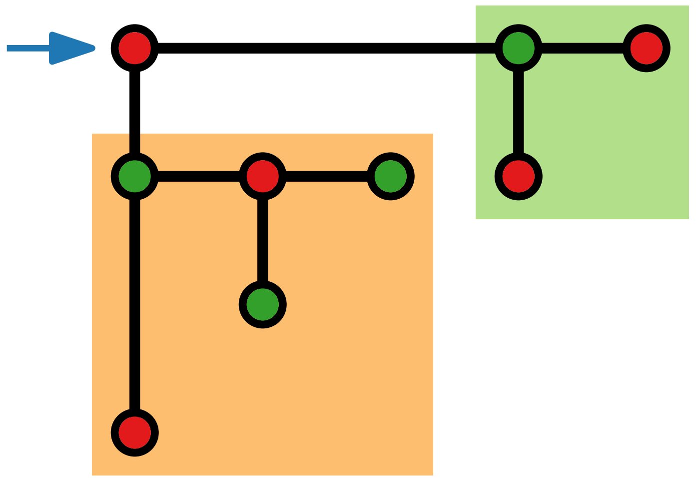


hv-drawing – balanced layout

Balanced approach

- Recursively compute layout for left and right subtrees
- Apply
  - horizontal combination if vertex is at odd depth
  - vertical combination if vertex is at even depth

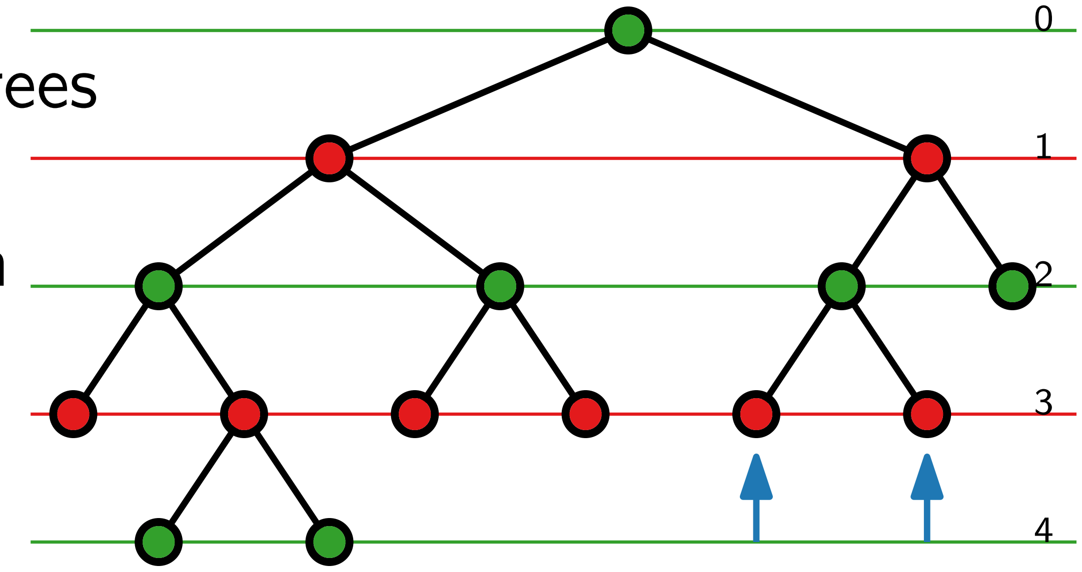


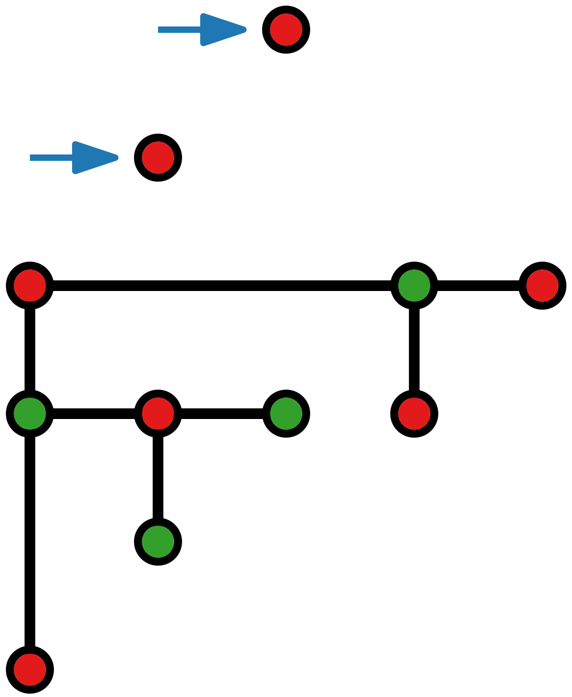


hv-drawing – balanced layout

Balanced approach

- Recursively compute layout for left and right subtrees
- Apply
  - horizontal combination if vertex is at odd depth
  - vertical combination if vertex is at even depth

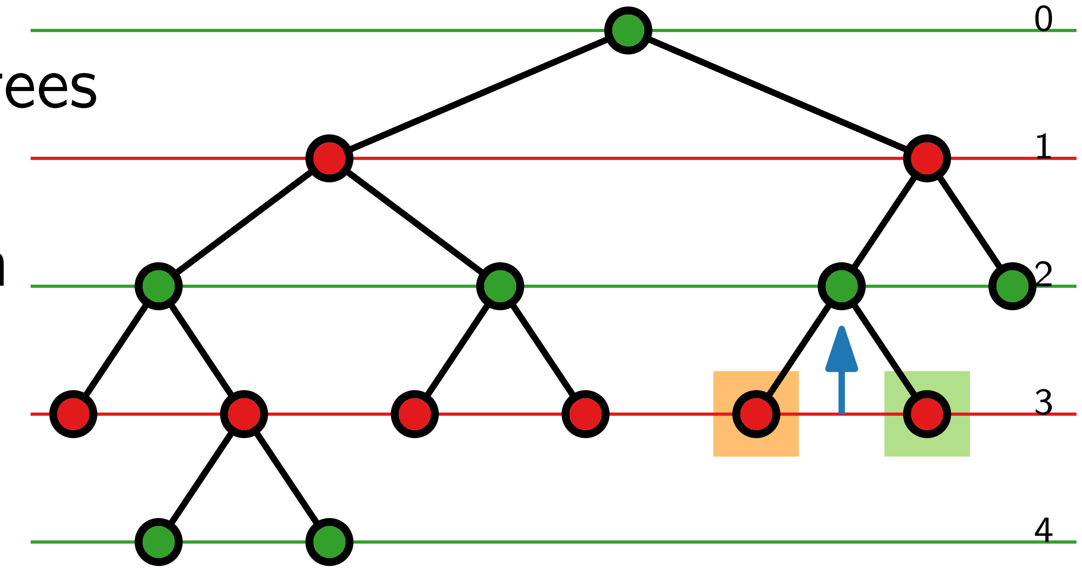


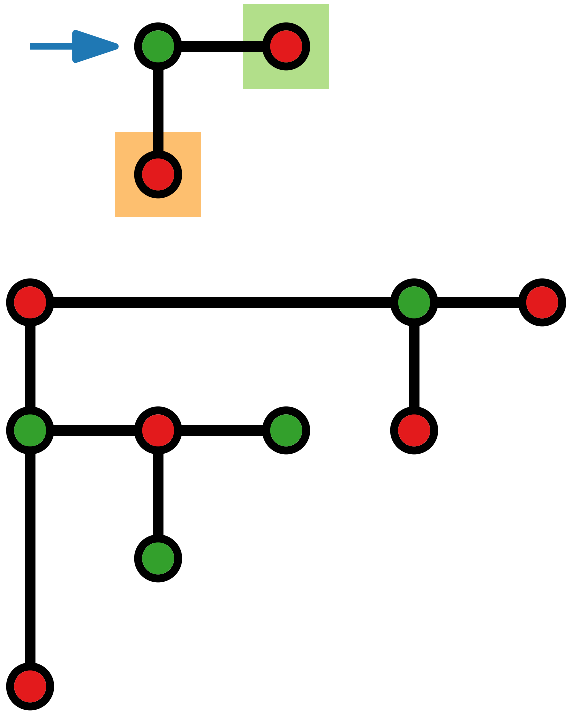


hv-drawing – balanced layout

Balanced approach

- Recursively compute layout for left and right subtrees
- Apply
  - horizontal combination if vertex is at odd depth
  - vertical combination if vertex is at even depth

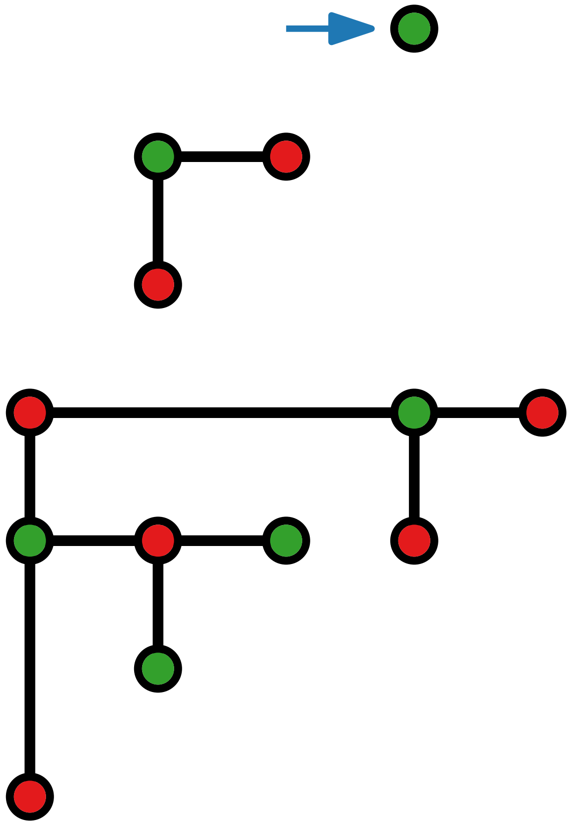


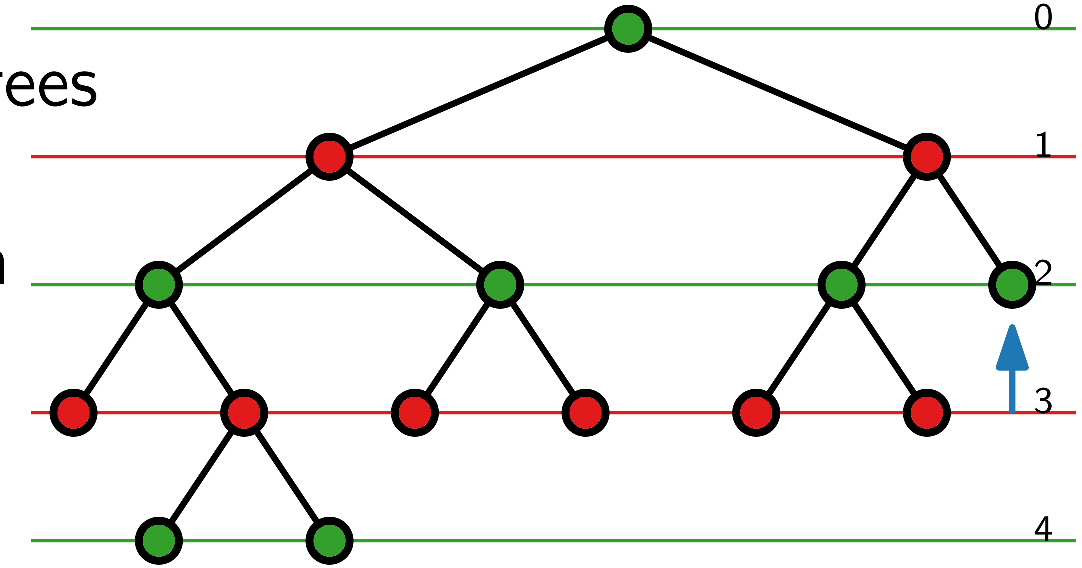


hv-drawing – balanced layout

Balanced approach

- Recursively compute layout for left and right subtrees
- Apply
  - horizontal combination if vertex is at odd depth
  - vertical combination if vertex is at even depth

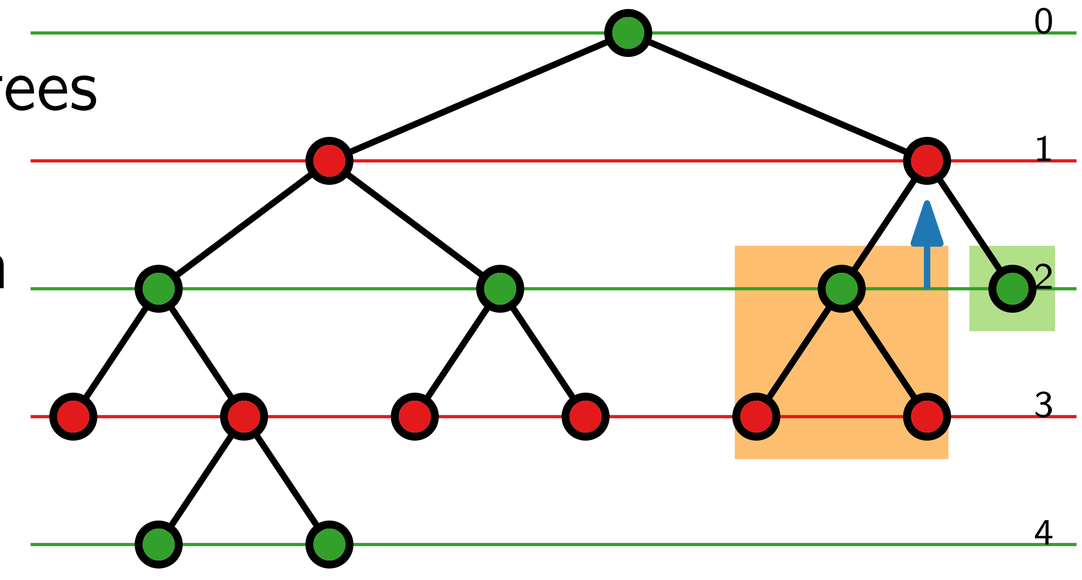


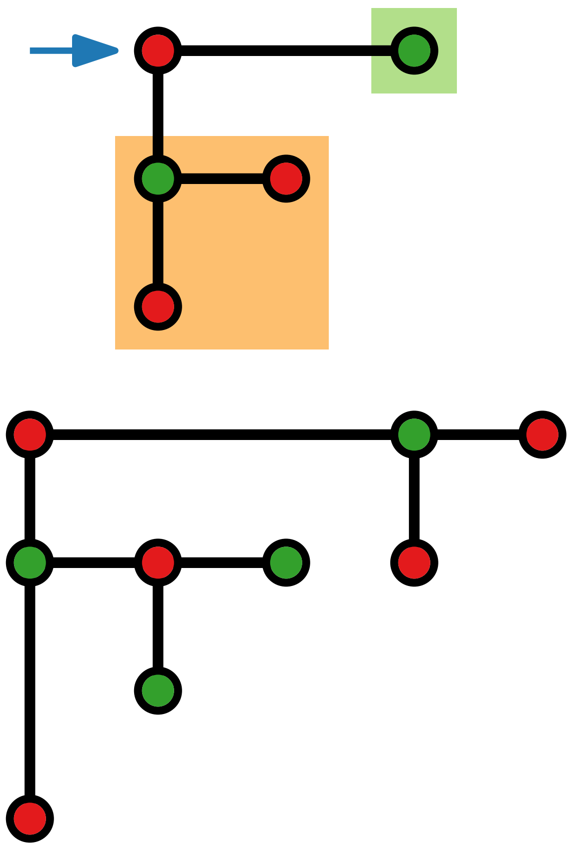


hv-drawing – balanced layout

Balanced approach

- Recursively compute layout for left and right subtrees
- Apply
  - horizontal combination if vertex is at odd depth
  - vertical combination if vertex is at even depth

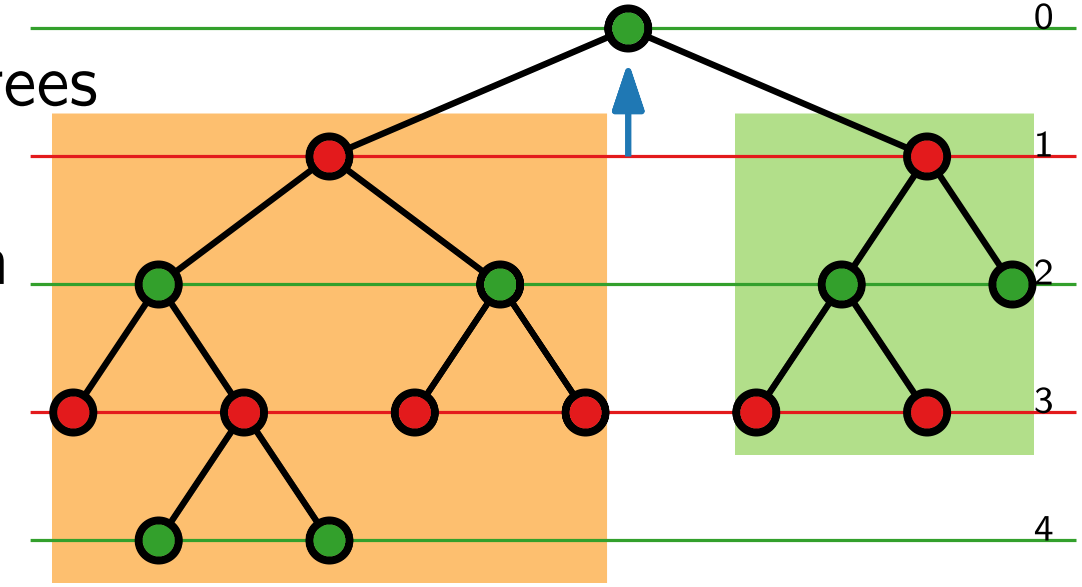


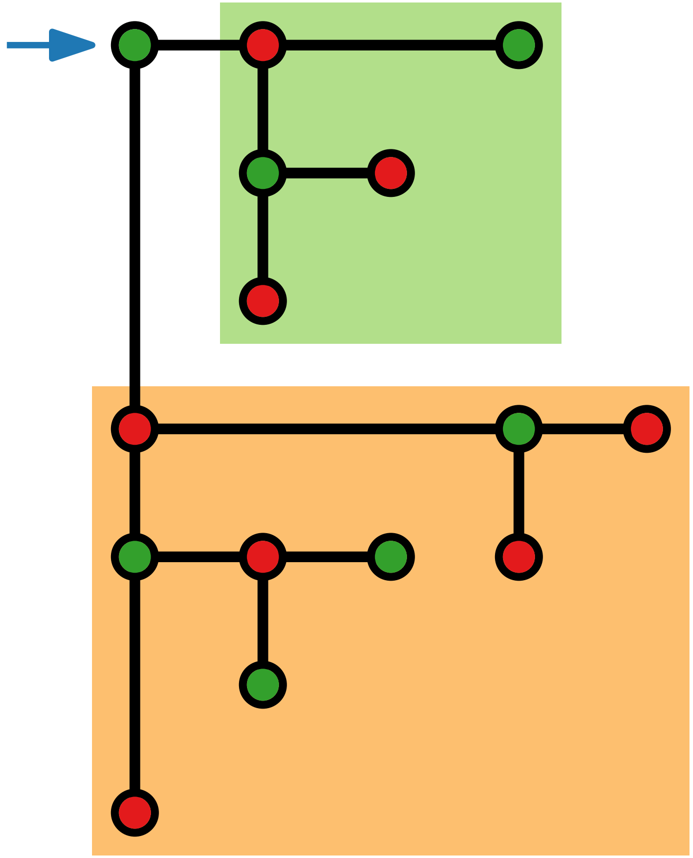


hv-drawing – balanced layout

Balanced approach

- Recursively compute layout for left and right subtrees
- Apply
  - horizontal combination if vertex is at odd depth
  - vertical combination if vertex is at even depth





hv-drawing – balanced layout

Lemma. Let T be a binary tree. The drawing constructed by balanced approach has

- area $\mathcal{O}(n)$ and
- constant aspect ratio

hv-drawing – balanced layout

Lemma. Let T be a binary tree. The drawing constructed by balanced approach has
- area $\mathcal{O}(n)$ and
- constant aspect ratio

Base case: $h = 0$

$W_0 = 0$, $H_0 = 0$

hv-drawing – balanced layout

Lemma. Let T be a binary tree. The drawing constructed by balanced approach has

- area $\mathcal{O}(n)$ and
- constant aspect ratio

Base case: $h = 0$

$W_0 = 0,\ H_0 = 0$

even height: $h = 2k$

$W_h,\ H_h$

hv-drawing – balanced layout

Lemma. Let T be a binary tree. The drawing constructed by balanced approach has
- area $\mathcal{O}(n)$ and
- constant aspect ratio

even height: $h = 2k$

W_h, H_h

- compute W_{h+1}, H_{h+1}

Base case: $h = 0$

$W_0 = 0,\ H_0 = 0$

hv-drawing – balanced layout

Lemma. Let T be a binary tree. The drawing constructed by balanced approach has
- area $\mathcal{O}(n)$ and
- constant aspect ratio

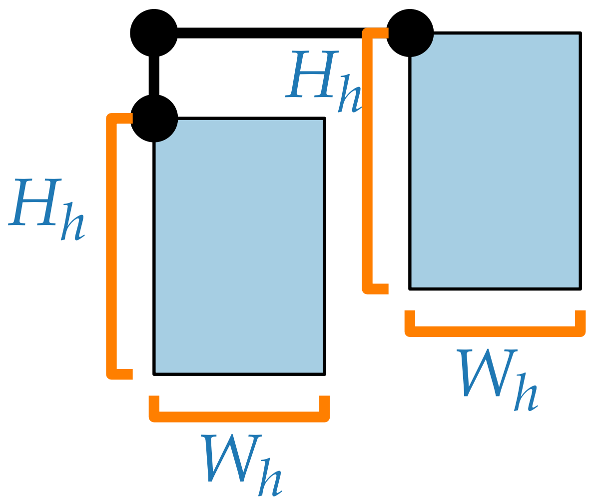
Base case: $h = 0$

$W_0 = 0$, $H_0 = 0$

even height: $h = 2k$

W_h, H_h

- compute W_{h+1}, H_{h+1}



hv-drawing – balanced layout

Lemma. Let T be a binary tree. The drawing constructed by balanced approach has
- area $\mathcal{O}(n)$ and
- constant aspect ratio

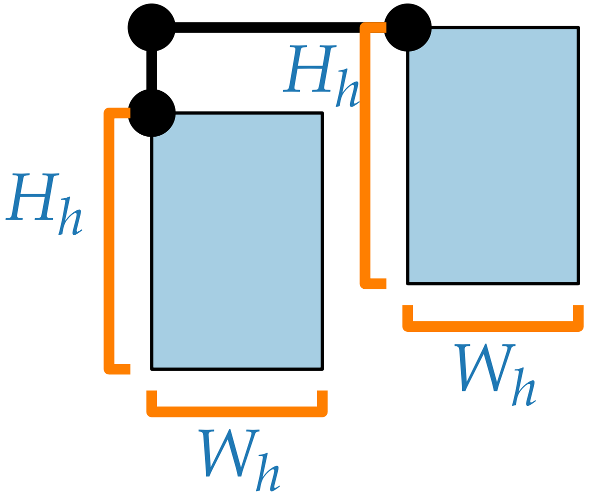
Base case: $h = 0$

$W_0 = 0$, $H_0 = 0$

even height: $h = 2k$

W_h, H_h

- compute W_{h+1}, H_{h+1}



$$W_{h+1} = 2W_h + 1$$
$$H_{h+1} = H_h + 1$$

hv-drawing – balanced layout

Lemma. Let T be a binary tree. The drawing constructed by balanced approach has
- area $\mathcal{O}(n)$ and
- constant aspect ratio

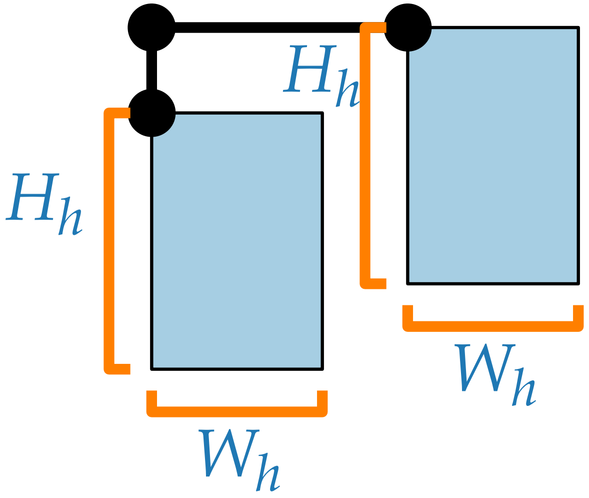
Base case: $h = 0$

$W_0 = 0,\ H_0 = 0$

even height: $h = 2k$

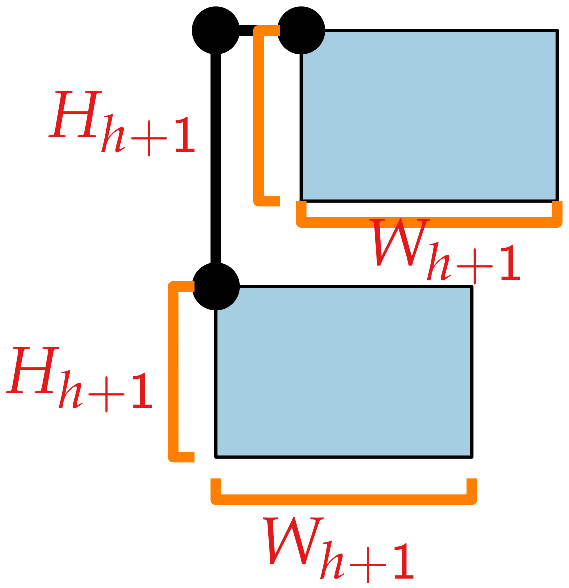
$W_h,\ H_h$

- compute $W_{h+1},\ H_{h+1}$



$$W_{h+1} = 2W_h + 1$$
$$H_{h+1} = H_h + 1$$

- compute $W_{h+2},\ H_{h+2}$



hv-drawing – balanced layout

Lemma. Let T be a binary tree. The drawing constructed by balanced approach has
- area $\mathcal{O}(n)$ and
- constant aspect ratio

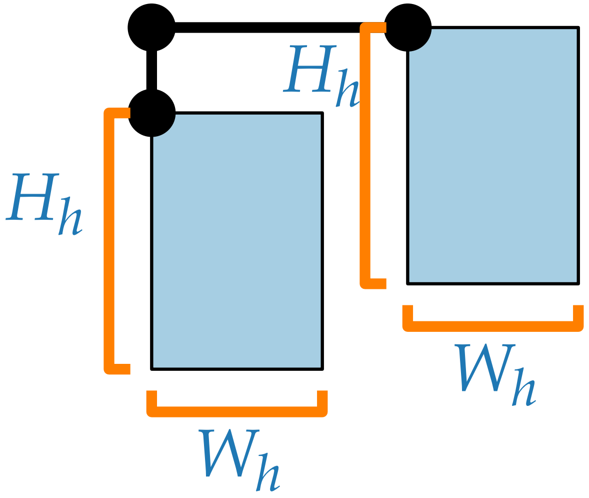
Base case: $h = 0$

$W_0 = 0,\ H_0 = 0$

even height: $h = 2k$

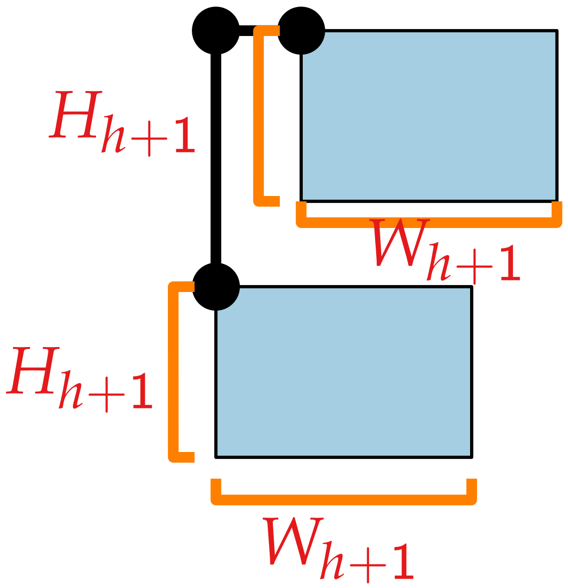
$W_h,\ H_h$

- compute $W_{h+1},\ H_{h+1}$



$$W_{h+1} = 2W_h + 1$$
$$H_{h+1} = H_h + 1$$

- compute $W_{h+2},\ H_{h+2}$



$$W_{h+2} = W_{h+1} + 1$$
$$H_{h+2} = 2H_{h+1} + 1$$

hv-drawing – balanced layout

Lemma. Let T be a binary tree. The drawing constructed by balanced approach has
- area $\mathcal{O}(n)$ and
- constant aspect ratio

Base case: $h = 0$

$W_0 = 0$, $H_0 = 0$

even height: $h = 2k$

W_h, H_h

$W_{h+2} = 2W_h + 2$

$H_{h+2} = 2H_h + 3$

hv-drawing – balanced layout

Lemma. Let T be a binary tree. The drawing constructed by balanced approach has
- area $\mathcal{O}(n)$ and
- constant aspect ratio

Base case: $h = 0$

$W_0 = 0,\ H_0 = 0$

even height: $h = 2k$

$W_h,\ H_h$

$$W_{h+2} = 2W_h + 2$$
$$H_{h+2} = 2H_h + 3$$

$$W_h = 2(2^{h/2} - 1)$$
$$H_h = 3(2^{h/2} - 1)$$

$$W_h = 2\sqrt{n} - 2$$
$$H_h = 3\sqrt{n} - 3$$

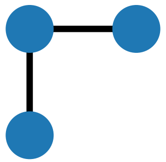
hv-drawing – balanced layout

Lemma. Let T be a binary tree. The drawing constructed by balanced approach has
- area $\mathcal{O}(n)$ and
- constant aspect ratio

Base case: $h = 0$

$W_0 = 0,\ H_0 = 0$

Base case: $h = 1$



$W_1 = 1,\ H_1 = 1$

even height: $h = 2k$

$W_h,\ H_h$

$$W_{h+2} = 2W_h + 2$$
$$H_{h+2} = 2H_h + 3$$

$$W_h = 2(2^{h/2} - 1) \quad \rightarrow \quad W_h = 2\sqrt{n} - 2$$
$$H_h = 3(2^{h/2} - 1) \quad \rightarrow \quad H_h = 3\sqrt{n} - 3$$

odd height: $h = 2k + 1$

$W_h,\ H_h$

$$W_{h+2} = 2W_h + 3 \quad \rightarrow \quad W_h = 2\sqrt{2n} - 3$$
$$H_{h+2} = 2H_h + 2 \quad \rightarrow \quad H_h = \tfrac{3}{2}\sqrt{2n} - 2$$

hv-drawing – result (2)

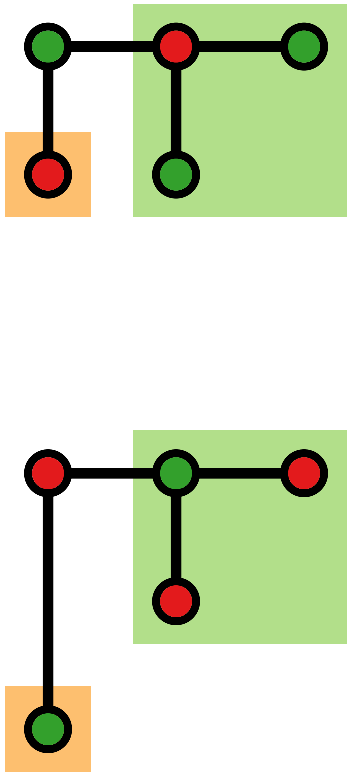
Theorem.
Let T be a binary tree with n vertices. The balanced algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is hv-drawing (planar, orthogonal)
- Width/Height is at most 2
- Area is in $\mathcal{O}(n)$

hv-drawing – result (2)

Theorem.
Let T be a binary tree with n vertices. The balanced algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

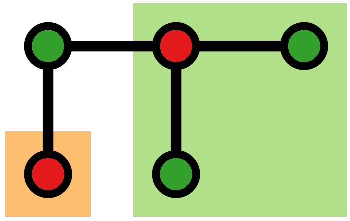
- Γ is hv-drawing (planar, orthogonal)
- Width/Height is at most 2
- Area is in $\mathcal{O}(n)$
- Isomorphic subtrees have congruent drawings up to translation only if the roots are both on odd or both on even depth.

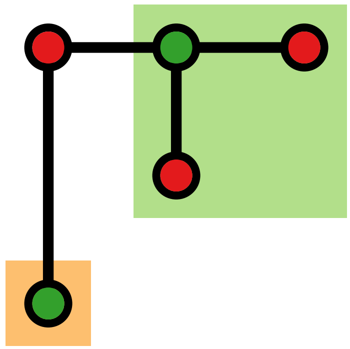


hv-drawing – result (2)

Theorem.
Let T be a binary tree with n vertices. The balanced algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is hv-drawing (planar, orthogonal)
- Width/Height is at most 2
- Area is in $\mathcal{O}(n)$
- Isomorphic subtrees have congruent drawings up to translation only if the roots are both on odd or both on even depth.



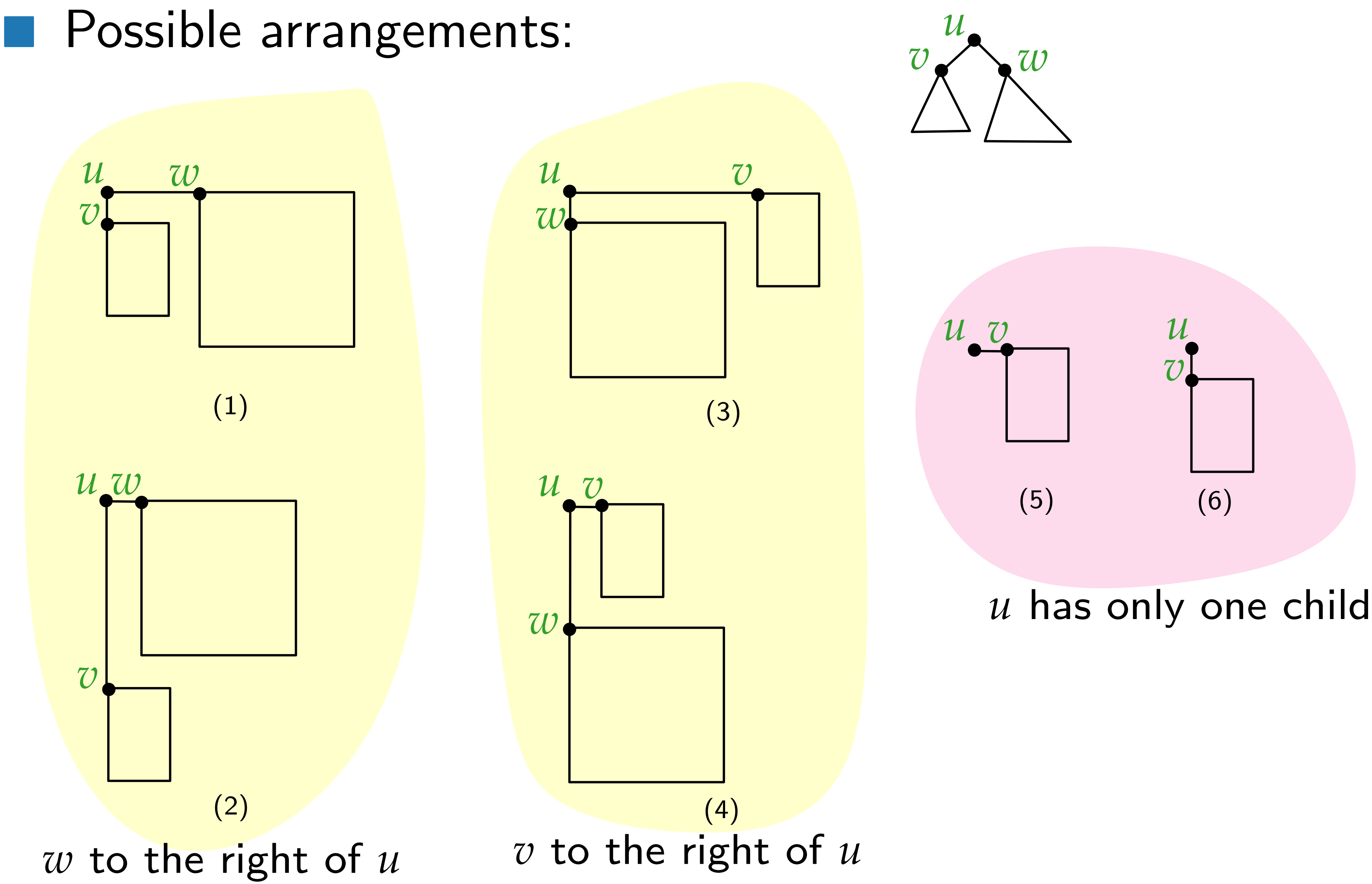


Optimal area?

- Not with divide & conquer approach, but
- can be computed with Dynamic Programming.

Optimum hv-layout for binary trees

- Possible arrangements:



Optimum hv-layout for binary trees

Algorithm Optimum_hv-layout
Input: Vertex v
Output: A list with all possible hv-layouts for T_v

If $h(T_v) == 0)$. –v is the only vertex in the tree
return trivial single vertex hv-layout
else

1. Build lists L_1 and L_2 of all possible hv-layouts of T_u^L and T_u^R, resp.
2. Combine L_1 and L_2 (by applying all possible arrangements) to build list L of all possible hv-layouts for T_v
3. **return** L

Optimum hv-layout for binary trees

Algorithm Optimum_hv-layout
Input: Vertex v
Output: A list with all possible hv-layouts for T_v

If $h(T_v) == 0)$. –v is the only vertex in the tree
return trivial single vertex hv-layout
else
1. Build lists L_1 and L_2 of all possible hv-layouts of T_u^L and T_u^R, resp.
2. Combine L_1 and L_2 (by applying all possible arrangements) to build list L of all possible hv-layouts for T_v
3. **return** L

- From the list at the root of the tree, select the optimum hv-layout.
  Optimum w.r.t.: area, perimeter, height, width, ...

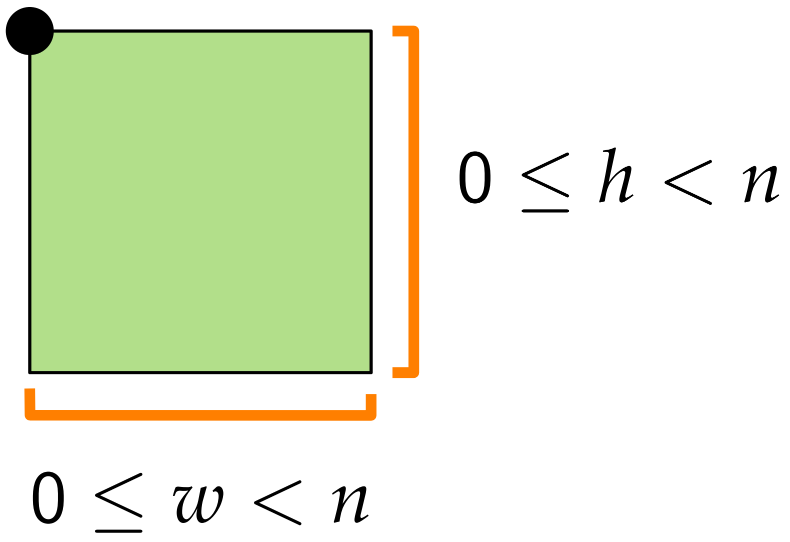
Optimum hv-layout for binary trees

Obervation 1: The number of possible hv-layouts is exponential

Optimum hv-layout for binary trees

Obervation 1: The number of possible hv-layouts is exponential

Obervation 2: The number of possible enclosing rectangles is at most n^2
[n possible different heights and n possible different widths]

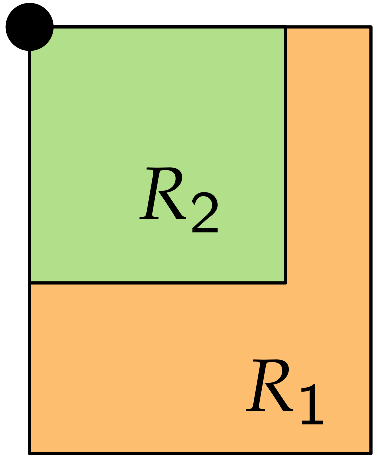


Optimum hv-layout for binary trees

Obervation 1: The number of possible hv-layouts is exponential

Observation 2: The number of possible enclosing rectangles is at most n^2
[n possible different heights and n possible different widths]

Observation 3: We only need to keep the enclosing rectangles that are not fully covered by other enclosing rectangles. We refer to them as *atoms*.



$$R_2 \subset R_1 \qquad \begin{cases} w_2 \leq w_1 \text{ and} \\ h_2 \leq h_1 \end{cases}$$

Optimum hv-layout for binary trees

Obervation 1: The number of possible hv-layouts is exponential

Obervation 2: The number of possible enclosing rectangles is at most n^2
[n possible different heights and n possible different widths]

Obervation 3: We only need to keep the enclosing rectangles that are not fully covered by other enclosing rectangles. We refer to them as *atoms*.

Lemma: For an n-vertex binary tree we have at most $n-1$ atoms.

Optimum hv-layout for binary trees

Obervation 1: The number of possible hv-layouts is exponential

Obervation 2: The number of possible enclosing rectangles is at most n^2
[n possible different heights and n possible different widths]

Obervation 3: We only need to keep the enclosing rectangles that are not fully covered by other enclosing rectangles. We refer to them as *atoms*.

Lemma: For an n-vertex binary tree we have at most $n-1$ atoms.

Proof: Observe that:

- Let each atom be of the form $[w \times h]$.
- There is only one atom for each w, $0 \leq w \leq n-1$. $\square$

Optimum hv-layout for binary trees

Time Analysis:

1. **Simple implementation:**
   - Combining the n^2 rectangles in each of L_1 and L_2 to get a list of n^4 rectangles.
     $\Rightarrow\ O(n^4)$ time
   - Remove duplicate rectangles $\Rightarrow\ O(n^4)$ time
   - Repeat for each internal tree node $\Rightarrow\ O(n \cdot n^4) = O(n^5)$ total time

Optimum hv-layout for binary trees

Time Analysis:

1. **Simple implementation:**
   - Combining the n^2 rectangles in each of L_1 and L_2 to get a list of n^4 rectangles.
     $\Rightarrow O(n^4)$ time
   - Remove duplicate rectangles $\Rightarrow O(n^4)$ time
   - Repeat for each internal tree node $\Rightarrow O(n \cdot n^4) = O(n^5)$ total time

2. **Implementation based on "atom-only" lists [Observation-3]**
   - Combine the n atoms in each of L_1 and L_2 and remove duplicates $\Rightarrow O(n^2)$ time
   - Repeat for each internal tree node $\Rightarrow O(n \cdot n^2) = O(n^3)$ total time

Optimum hv-layout for binary trees

Time Analysis:

1. **Simple implementation:**
   - Combining the n^2 rectangles in each of L_1 and L_2 to get a list of n^4 rectangles. $\Rightarrow\ O(n^4)$ time
   - Remove duplicate rectangles $\Rightarrow\ O(n^4)$ time
   - Repeat for each internal tree node $\Rightarrow\ O(n \cdot n^4) = O(n^5)$ total time

2. **Implementation based on "atom-only" lists [Observation-3]**
   - Combine the n atoms in each of L_1 and L_2 and remove duplicates $\Rightarrow\ O(n^2)$ time
   - Repeat for each internal tree node $\Rightarrow\ O(n \cdot n^2) = O(n^3)$ total time

3. **Fast "atom-based" implementation**
   - Combine the n atoms in each of L_1 and L_2 and remove duplicates by a "merge-like" operation $\Rightarrow\ O(n)$ time
   - Repeat for each internal tree node $\Rightarrow\ O(n \cdot n) = O(n^2)$ total time

Optimum hv-layout for binary trees

Time Analysis:

2. **Implementation based on "atom-only" lists [Observation-3]**
   - Combine the n atoms in each of L_1 and L_2 and remove duplicates $\Rightarrow$ $O(n^2)$ time
   - Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n^2) = O(n^3)$ total time

Optimum hv-layout for binary trees

Time Analysis:

2. **Implementation based on "atom-only" lists [Observation-3]**
   - Combine the n atoms in each of L_1 and L_2 and remove duplicates $\Rightarrow$ $O(n^2)$ time
   - Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n^2) = O(n^3)$ total time

atoms: array of length n

atoms$[i]$ $=$ atom with length i

- for each combination of L_1 and L_2 update array of atoms

Optimum hv-layout for binary trees

Time Analysis:

2. **Implementation based on "atom-only" lists [Observation-3]**

- Combine the n atoms in each of L_1 and L_2 and remove duplicates $\Rightarrow$ $O(n^2)$ time
- Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n^2) = O(n^3)$ total time

atoms: array of length n

atoms$[i]$ = atom with length i

- for each combination of L_1 and L_2 update array of atoms

Obervation: width is increasing $w_i < w_j$
height is decreasing $h_i > h_j$

Optimum hv-layout for binary trees

Time Analysis:

3. **Fast "atom-based" implementation**

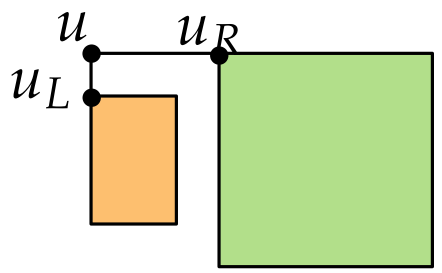
- Combine the n atoms in each of L_1 and L_2 and remove duplicates by a "merge-like" operation $\Rightarrow$ $O(n)$ time
- Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n) = O(n^2)$ total time

Optimum hv-layout for binary trees

Time Analysis:

3. **Fast "atom-based" implementation**

- Combine the n atoms in each of L_1 and L_2 and remove duplicates by a "merge-like" operation $\Rightarrow$ $O(n)$ time
- Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n) = O(n^2)$ total time



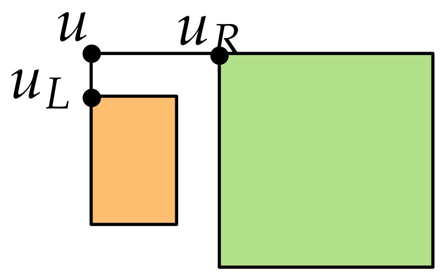
a_L: $\{p_0, \ldots, p_k\}$, $p_i = (w_i, h_i)$
a_R: $\{q_0, \ldots, q_\ell\}$, $q_j = (w'_j, h'_j)$

Optimum hv-layout for binary trees

Time Analysis:

3. **Fast "atom-based" implementation**

- Combine the n atoms in each of L_1 and L_2 and remove duplicates by a "merge-like" operation $\Rightarrow$ $O(n)$ time
- Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n) = O(n^2)$ total time



a_L: $\{p_0, \ldots, p_k\}$, $p_i = (w_i, h_i)$
a_R: $\{q_0, \ldots, q_\ell\}$, $q_j = (w'_j, h'_j)$

combination $c(p_i, q_j)$:

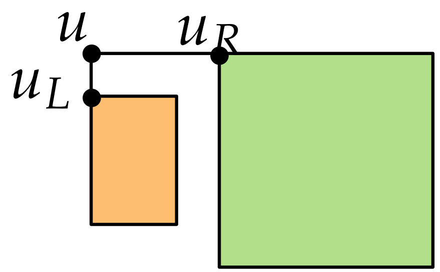
- $W = w_i + w'_j + 1$
- $H = \max\{h_i + 1, h'_j\}$

Optimum hv-layout for binary trees

Time Analysis:

3. **Fast "atom-based" implementation**

- Combine the n atoms in each of L_1 and L_2 and remove duplicates by a "merge-like" operation $\Rightarrow$ $O(n)$ time
- Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n) = O(n^2)$ total time



a_L: $\{p_0, \ldots, p_k\}$, $p_i = (w_i, h_i)$

a_R: $\{q_0, \ldots, q_\ell\}$, $q_j = (w'_j, h'_j)$

combination $c(p_i, q_j)$:

- $W = w_i + w'_j + 1$
- $H = \max\{h_i + 1, h'_j\}$

For fixed $p_i = (w_i, h_i)$

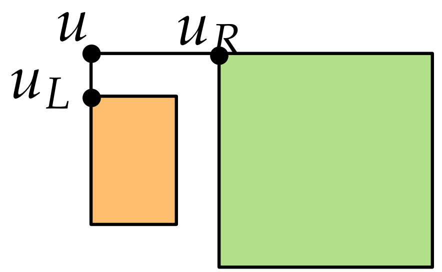
- W is increasing
- $H = \begin{cases} h'_j, \text{ for } h'_j > h_i + 1 \\ h_i, \text{ for } h'_j \leq h_i + 1 \end{cases}$

Optimum hv-layout for binary trees

Time Analysis:

3. **Fast "atom-based" implementation**

- Combine the n atoms in each of L_1 and L_2 and remove duplicates by a "merge-like" operation $\Rightarrow$ $O(n)$ time
- Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n) = O(n^2)$ total time



a_L: $\{p_0, \ldots, p_k\}$, $p_i = (w_i, h_i)$

a_R: $\{q_0, \ldots, q_\ell\}$, $q_j = (w'_j, h'_j)$

combination $c(p_i, q_j)$:

- $W = w_i + w'_j + 1$
- $H = \max\{h_i + 1, h'_j\}$

For fixed $p_i = (w_i, h_i)$

- W is increasing
- $H = \begin{cases} h'_j, & \text{for } h'_j > h_i + 1 \\ h_i, & \text{for } h'_j \leq h_i + 1 \end{cases}$

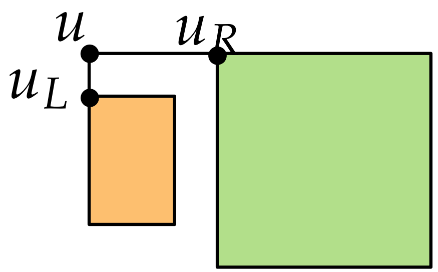
enclosed !!

Optimum hv-layout for binary trees

Time Analysis:

3. **Fast "atom-based" implementation**

- Combine the n atoms in each of L_1 and L_2 and remove duplicates by a "merge-like" operation $\Rightarrow$ $O(n)$ time
- Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n) = O(n^2)$ total time



a_L: $\{p_0, \ldots, p_k\}$, $p_i = (w_i, h_i)$
a_R: $\{q_0, \ldots, q_\ell\}$, $q_j = (w'_j, h'_j)$

combination $c(p_i, q_j)$:

- $W = w_i + w'_j + 1$
- $H = \max\{h_i + 1, h'_j\}$

For fixed $p_i = (w_i, h_i)$

- There exists smallest $j(i)$ s.t. $h'_{j(i)} \leq h_i + 1$
- atoms defined only for $j \leq j(i)$
- $j(i)$ is increasing
- $c(p_{i'>i}, q_j)$ enclosed by $c(p_i, q_j)$ for $j \leq j(i)$

Optimum hv-layout for binary trees

Time Analysis:

3. **Fast "atom-based" implementation**

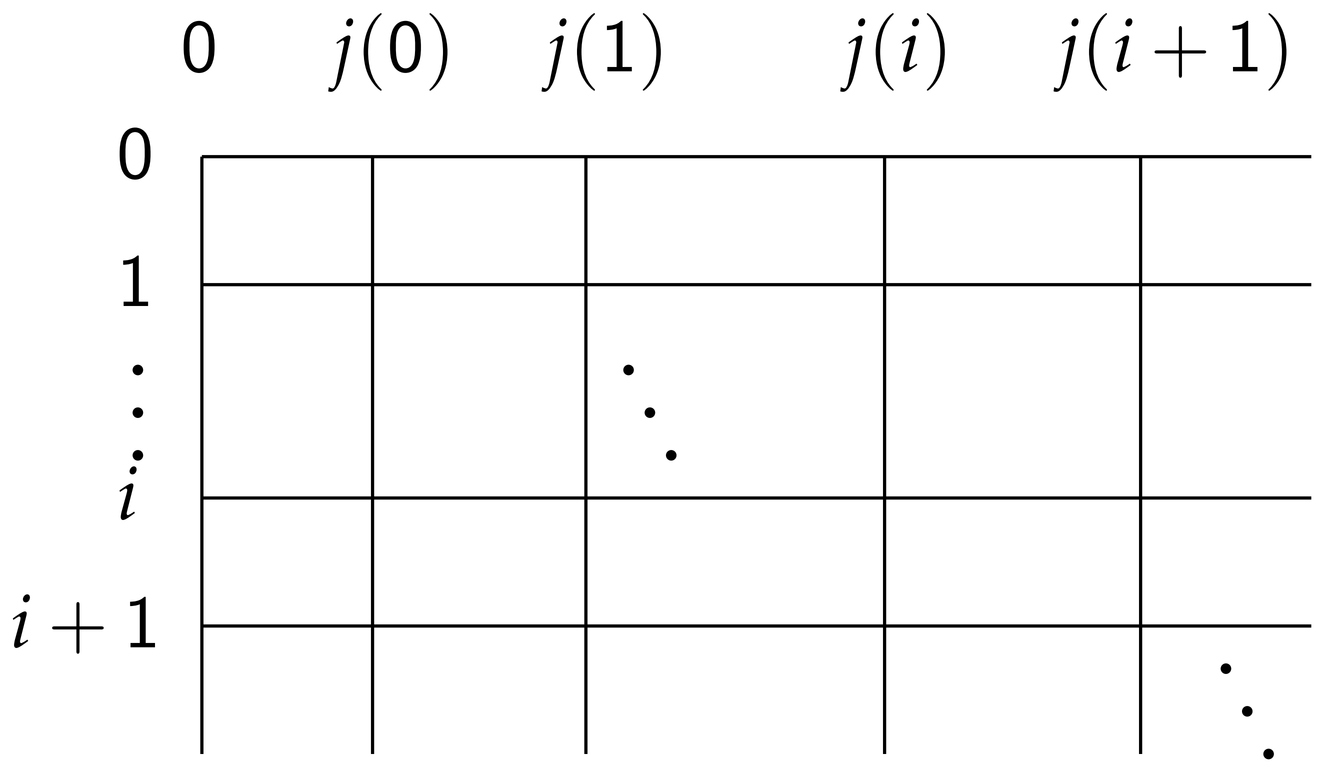
- Combine the n atoms in each of L_1 and L_2 and remove duplicates by a "merge-like" operation $\Rightarrow$ $O(n)$ time
- Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n) = O(n^2)$ total time

Optimum hv-layout for binary trees

Time Analysis:

3. **Fast “atom-based” implementation**

- Combine the n atoms in each of L_1 and L_2 and remove duplicates by a “merge-like” operation $\Rightarrow$ $O(n)$ time
- Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n) = O(n^2)$ total time

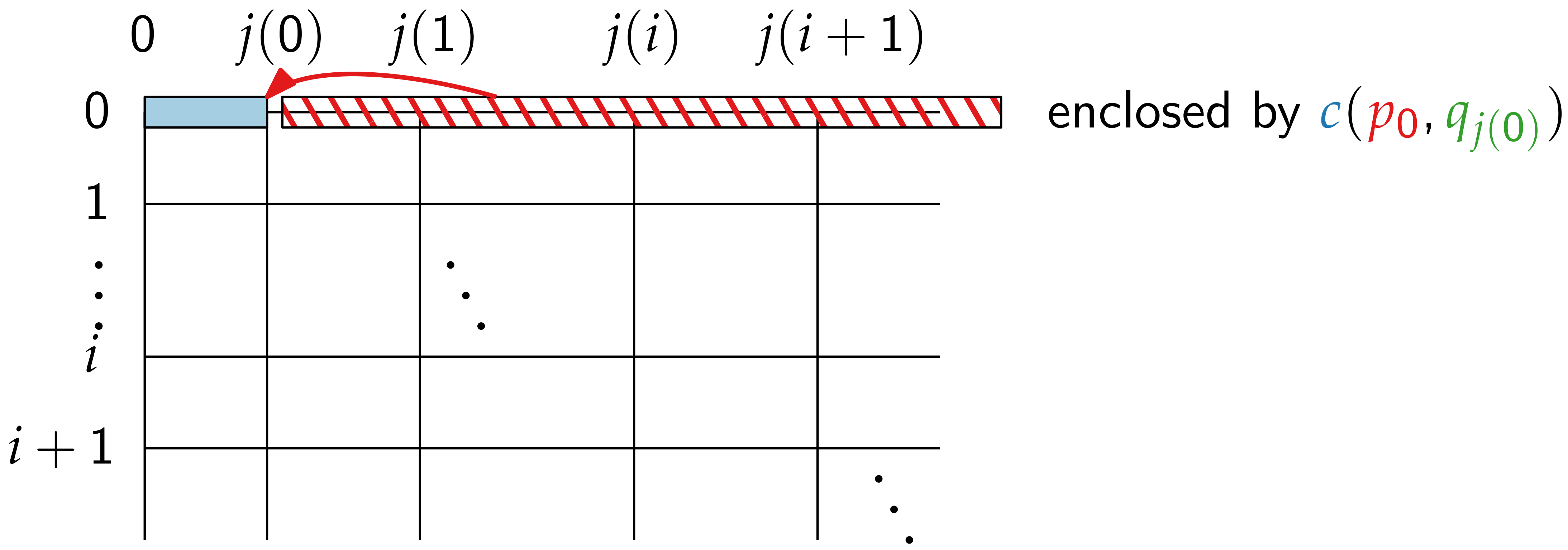


Optimum hv-layout for binary trees

Time Analysis:

3. **Fast "atom-based" implementation**

- Combine the n atoms in each of L_1 and L_2 and remove duplicates by a "merge-like" operation $\Rightarrow$ $O(n)$ time
- Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n) = O(n^2)$ total time

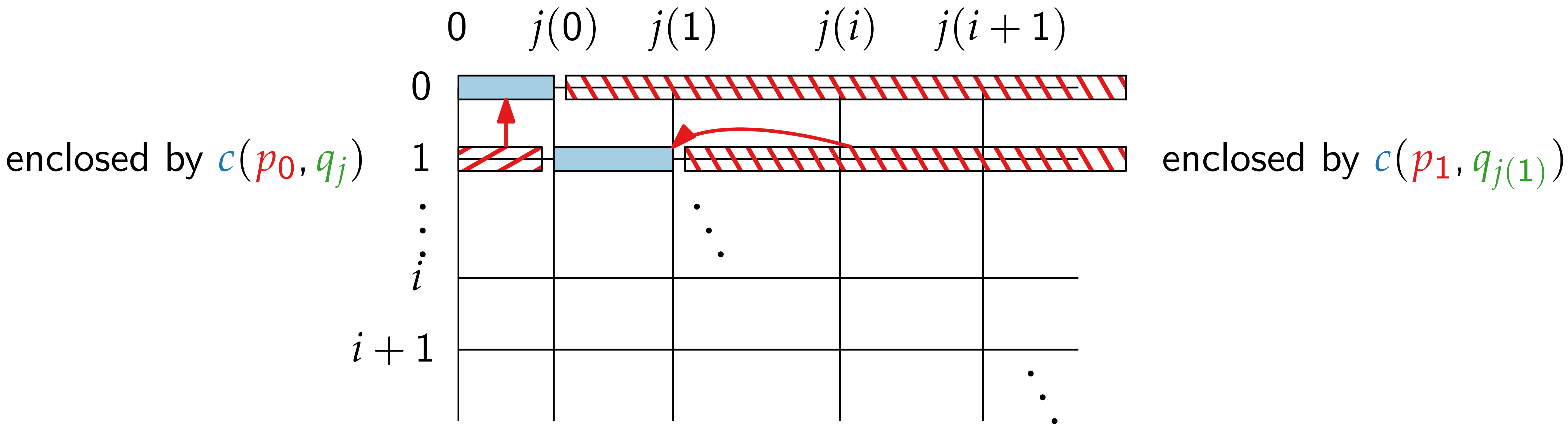


Optimum hv-layout for binary trees

Time Analysis:

3. **Fast “atom-based” implementation**

- Combine the n atoms in each of L_1 and L_2 and remove duplicates by a “merge-like” operation $\Rightarrow$ $O(n)$ time
- Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n) = O(n^2)$ total time

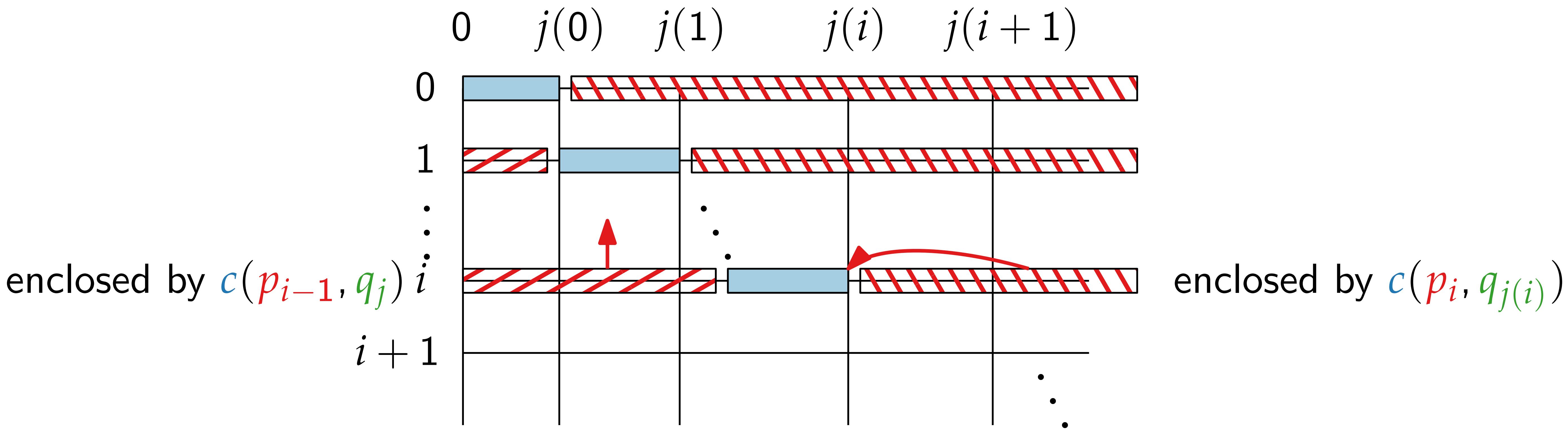


Optimum hv-layout for binary trees

Time Analysis:

3. **Fast "atom-based" implementation**

- Combine the n atoms in each of L_1 and L_2 and remove duplicates by a "merge-like" operation $\Rightarrow$ $O(n)$ time
- Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n) = O(n^2)$ total time

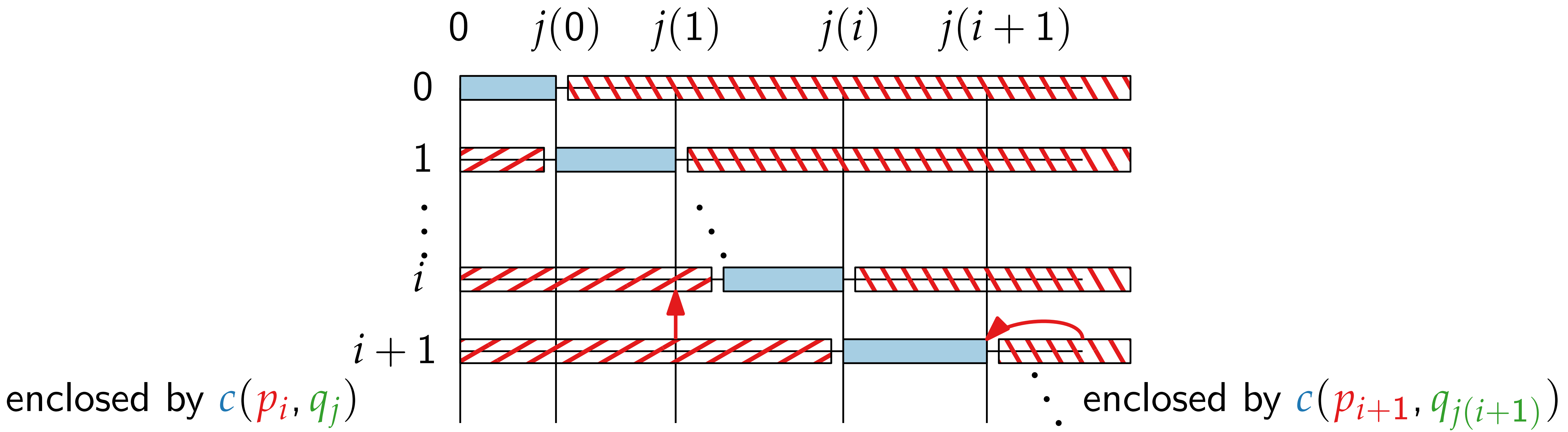


Optimum hv-layout for binary trees

Time Analysis:

3. **Fast "atom-based" implementation**

- Combine the n atoms in each of L_1 and L_2 and remove duplicates by a "merge-like" operation $\Rightarrow$ $O(n)$ time
- Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n) = O(n^2)$ total time

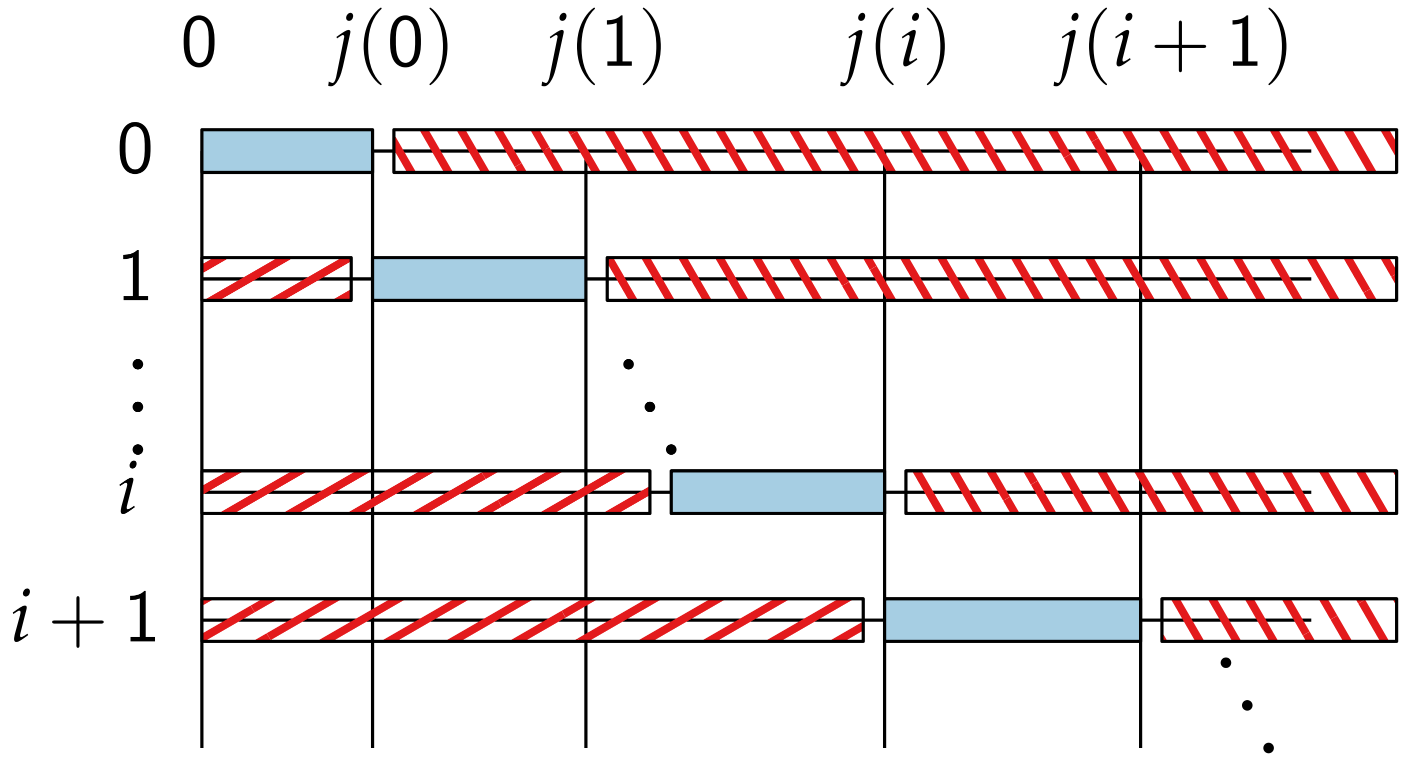


Optimum hv-layout for binary trees

Time Analysis:

3. **Fast "atom-based" implementation**

- Combine the n atoms in each of L_1 and L_2 and remove duplicates by a "merge-like" operation $\Rightarrow$ $O(n)$ time
- Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n) = O(n^2)$ total time



Optimum hv-layout for binary trees

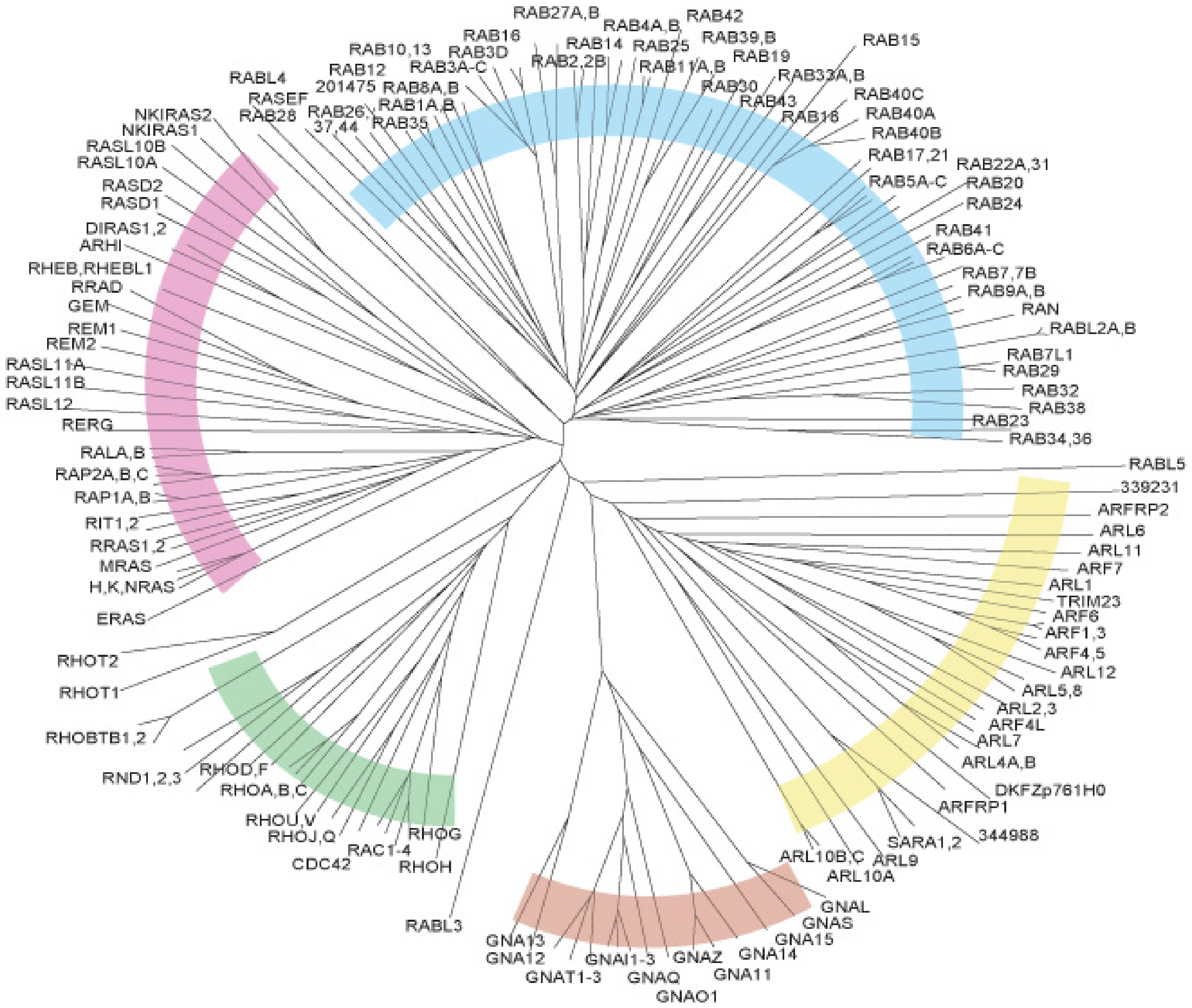
Time Analysis:

3. **Fast "atom-based" implementation**

- Combine the n atoms in each of L_1 and L_2 and remove duplicates by a "merge-like" operation $\Rightarrow$ $O(n)$ time
- Repeat for each internal tree node $\Rightarrow$ $O(n \cdot n) = O(n^2)$ total time

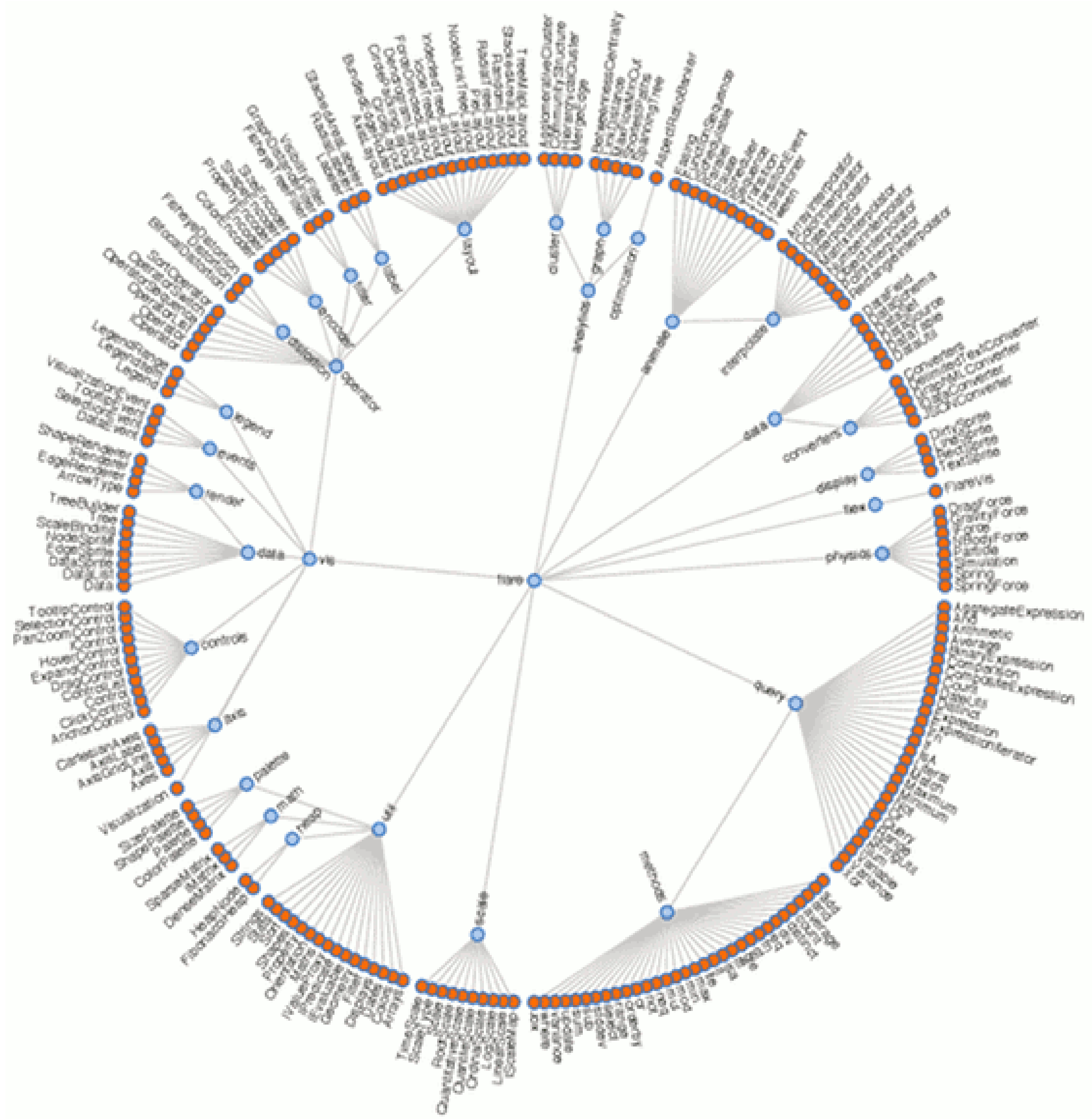
```
combine1(atoms a_L, atoms a_R)
    i ← 0
    j ← 0
    while i ≤ k and j ≤ ℓ do
        compute combination
        if h'_j > h_i + 1 then
            j ← j + 1
        else
            i ← i + 1
```

Radial layout – applications

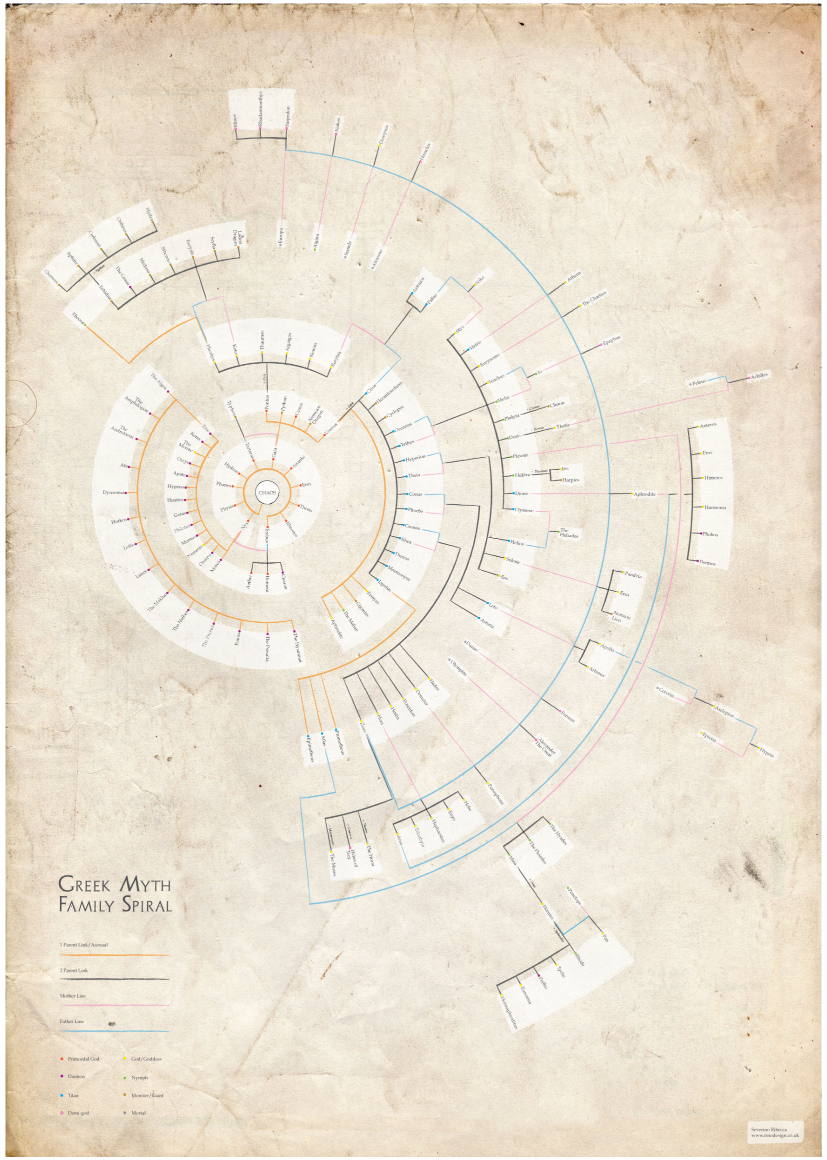


Phylogenetic tree
by Colicelli, ScienceSignaling, 2004

Radial layout – applications

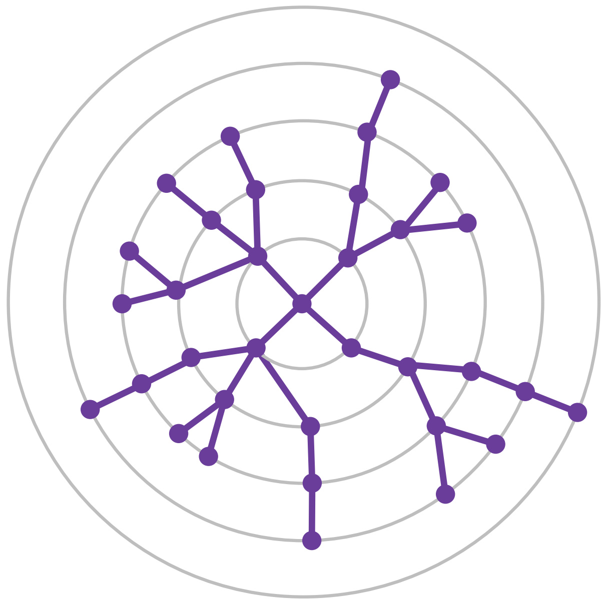


Flare Visualization Toolkit code structure
by Heer, Bostock and Ogievetsky, 2010



Greek Myth Family
by Ribecca, 2011

Radial layout – drawing style



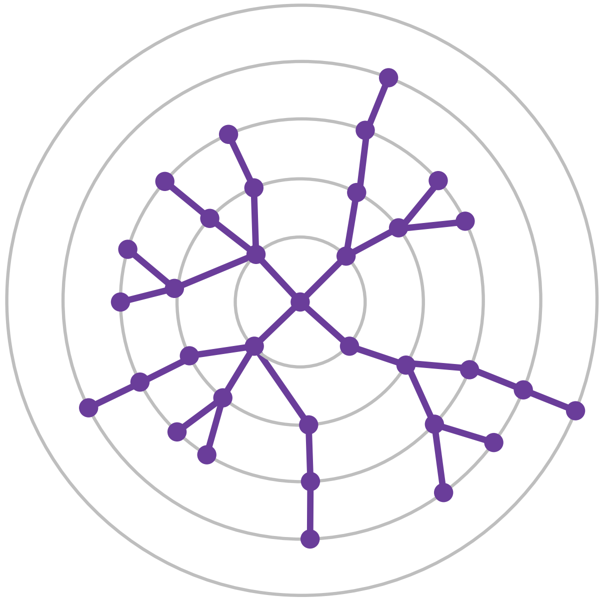
Drawing conventions

- Vertices lie on circular layers according to their depth
- Drawing is planar

Drawing aesthetics

- Distribution of the vertices

Radial layout – drawing style



Drawing conventions

- Vertices lie on circular layers according to their depth
- Drawing is planar

Drawing aesthetics

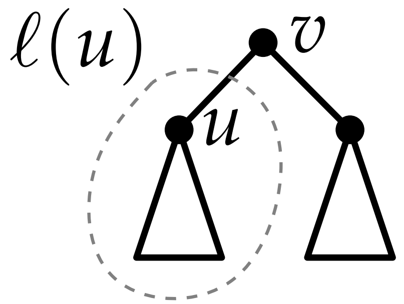
- Distribution of the vertices

How may an algorithm optimise the distribution of the vertices?

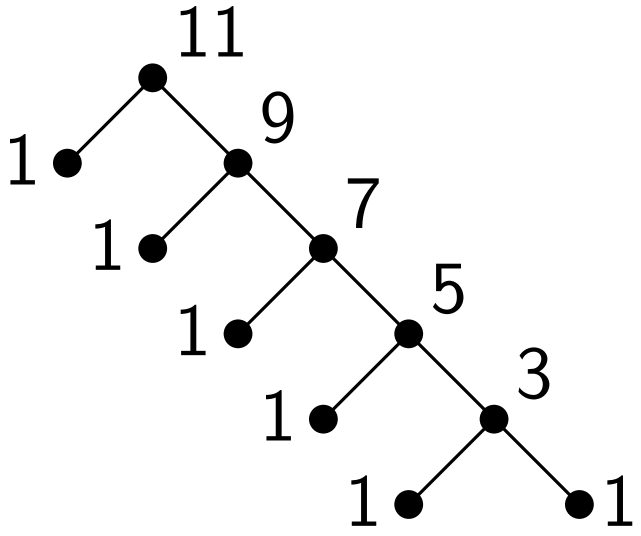
Radial layout – algorithm attempt

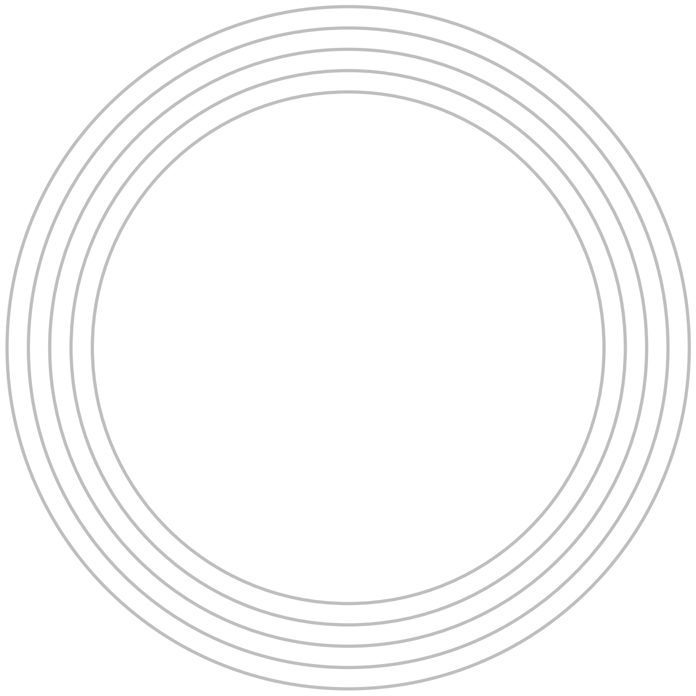
Idea

- Angle corresponding to size $\ell(u)$ of $T(u)$:



$$\tau_u = \frac{\ell(u)}{\ell(v) - 1} \tau_v$$

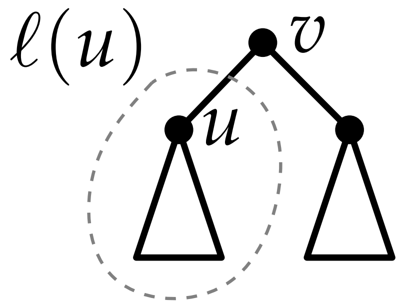




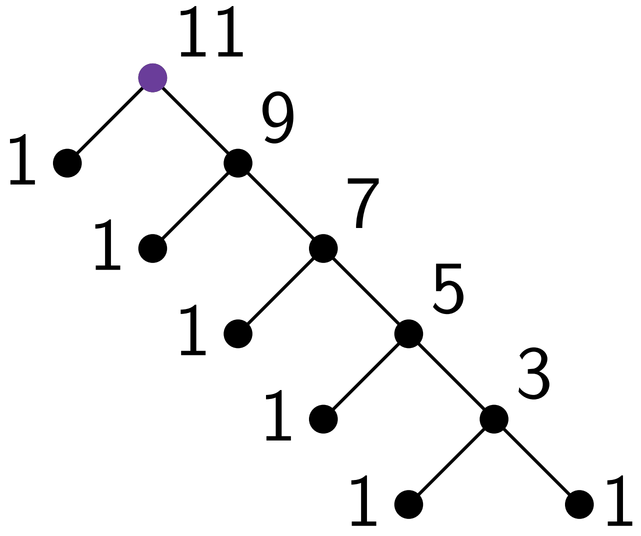
Radial layout – algorithm attempt

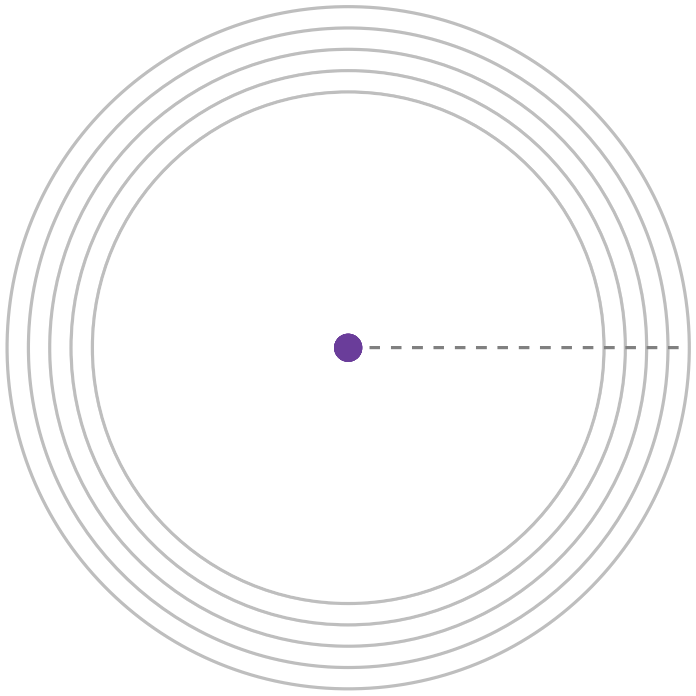
Idea

- Angle corresponding to size $\ell(u)$ of $T(u)$:



$$\tau_u = \frac{\ell(u)}{\ell(v) - 1} \tau_v$$

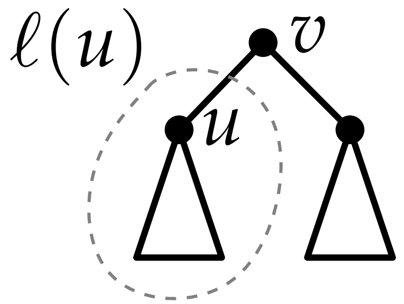




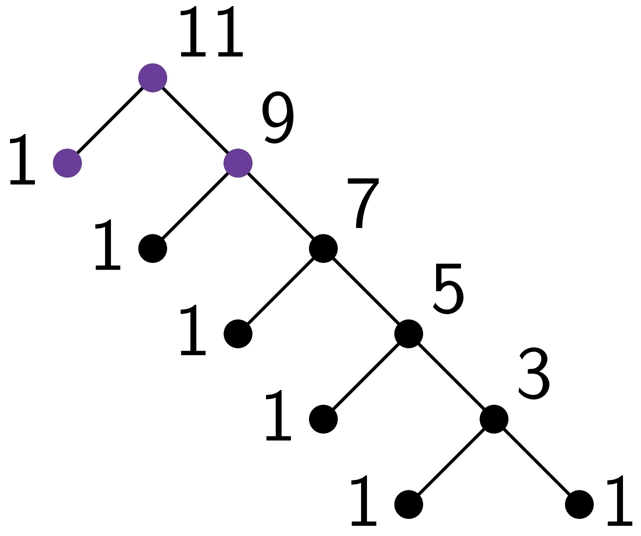
Radial layout – algorithm attempt

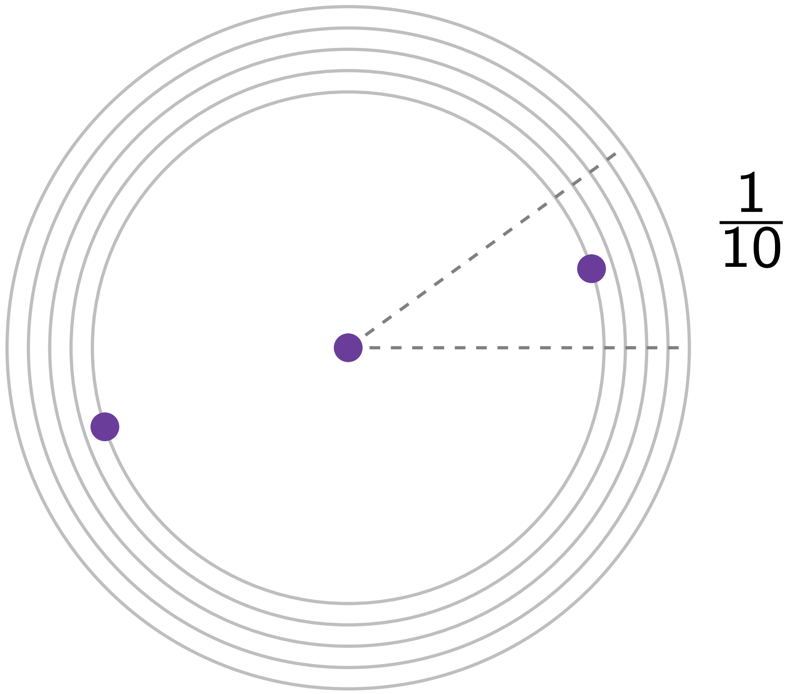
Idea

- Angle corresponding to size $\ell(u)$ of $T(u)$:



$$\tau_u = \frac{\ell(u)}{\ell(v) - 1}\tau_v$$

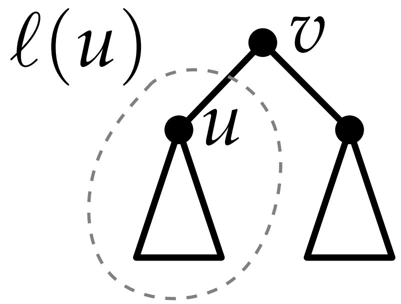




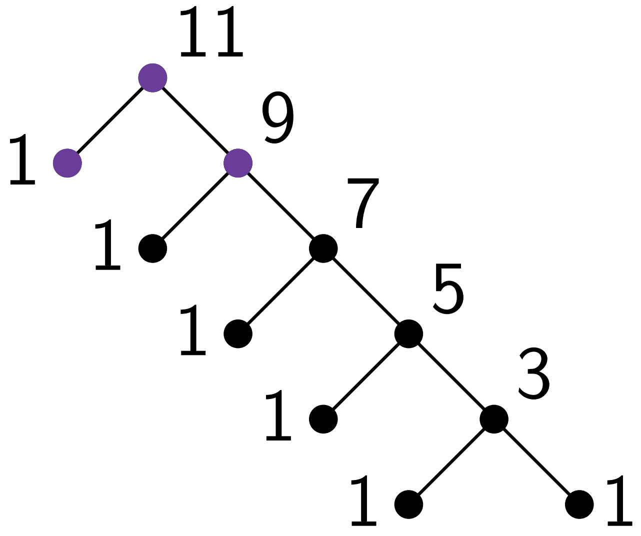
Radial layout – algorithm attempt

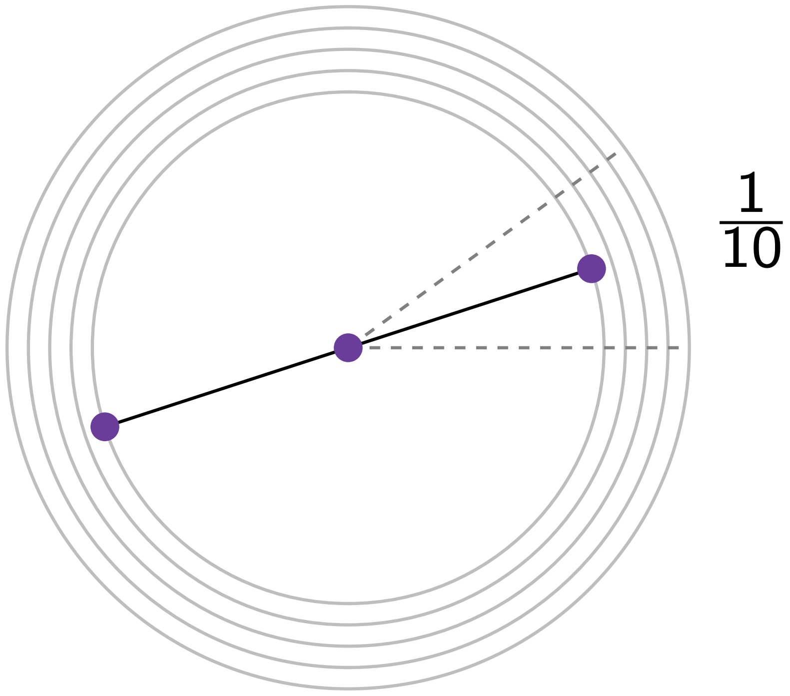
Idea

- Angle corresponding to size $\ell(u)$ of $T(u)$:



$$\tau_u = \frac{\ell(u)}{\ell(v) - 1}\tau_v$$



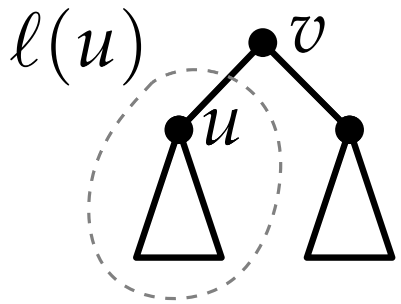


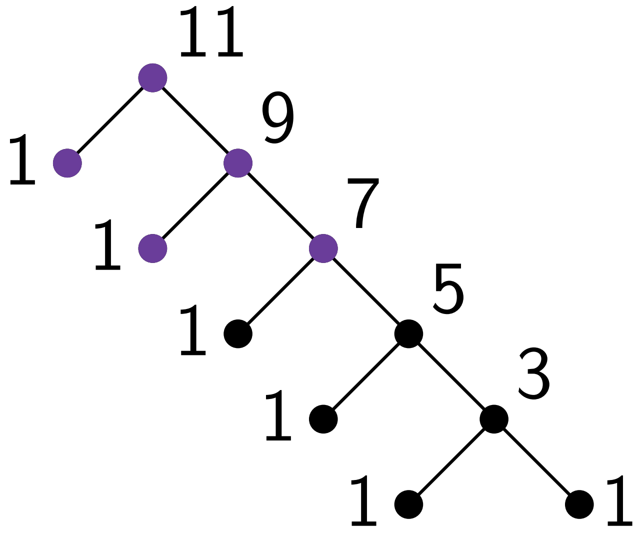
Radial layout – algorithm attempt

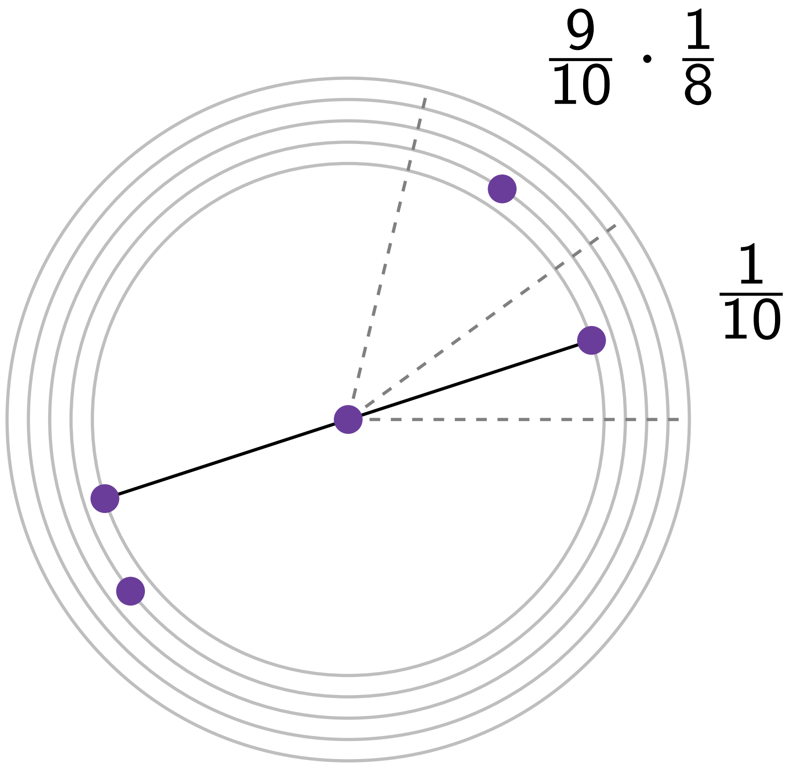
Idea

- Angle corresponding to size $\ell(u)$ of $T(u)$:

$$\tau_u = \frac{\ell(u)}{\ell(v) - 1}\tau_v$$



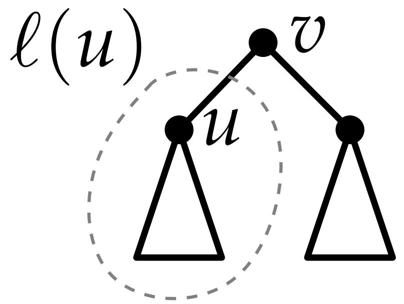




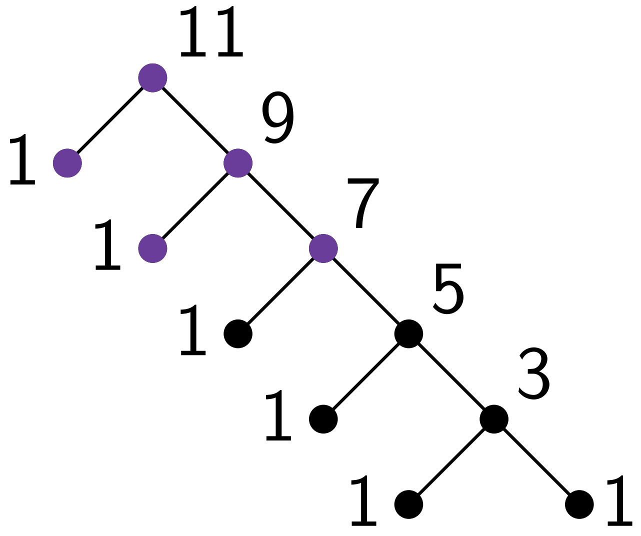
Radial layout – algorithm attempt

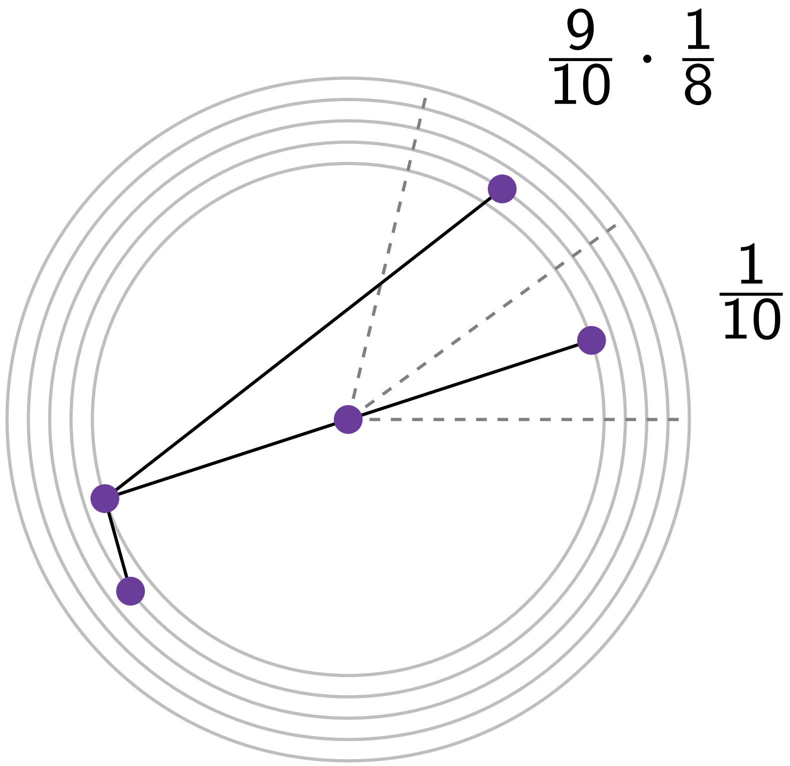
Idea

- Angle corresponding to size $\ell(u)$ of $T(u)$:



$$\tau_u = \frac{\ell(u)}{\ell(v) - 1} \tau_v$$

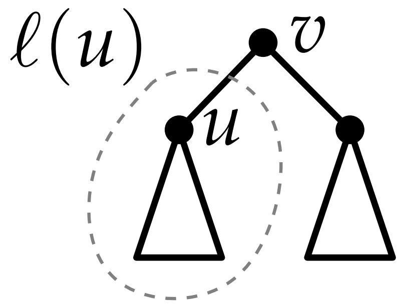




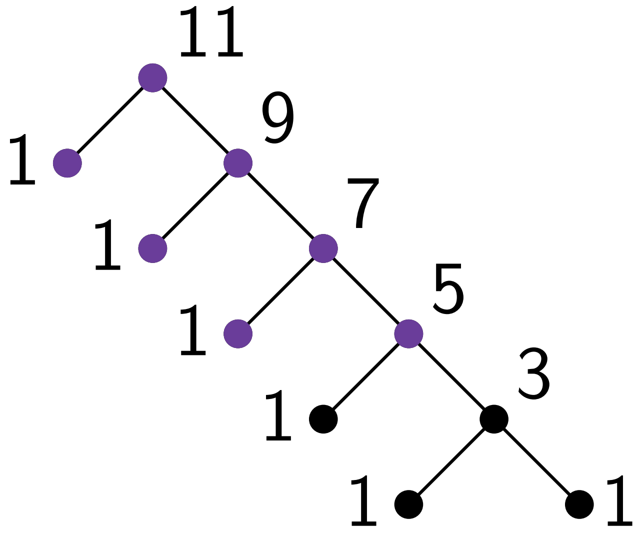
Radial layout – algorithm attempt

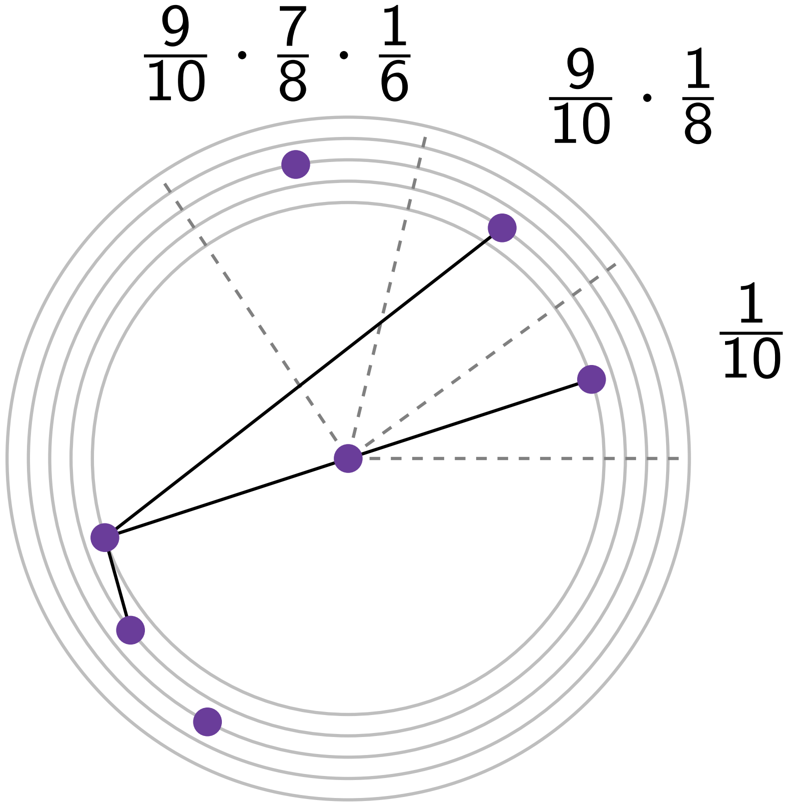
Idea

- Angle corresponding to size $\ell(u)$ of $T(u)$:



$$\tau_u = \frac{\ell(u)}{\ell(v) - 1} \tau_v$$

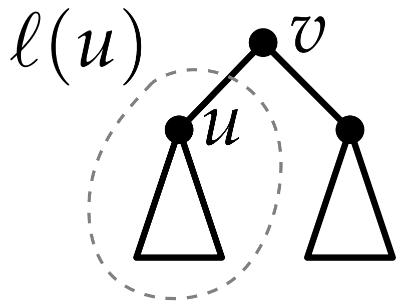




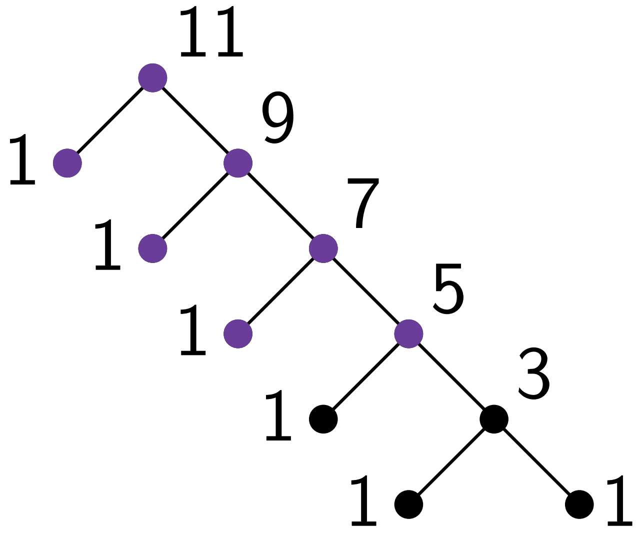
Radial layout – algorithm attempt

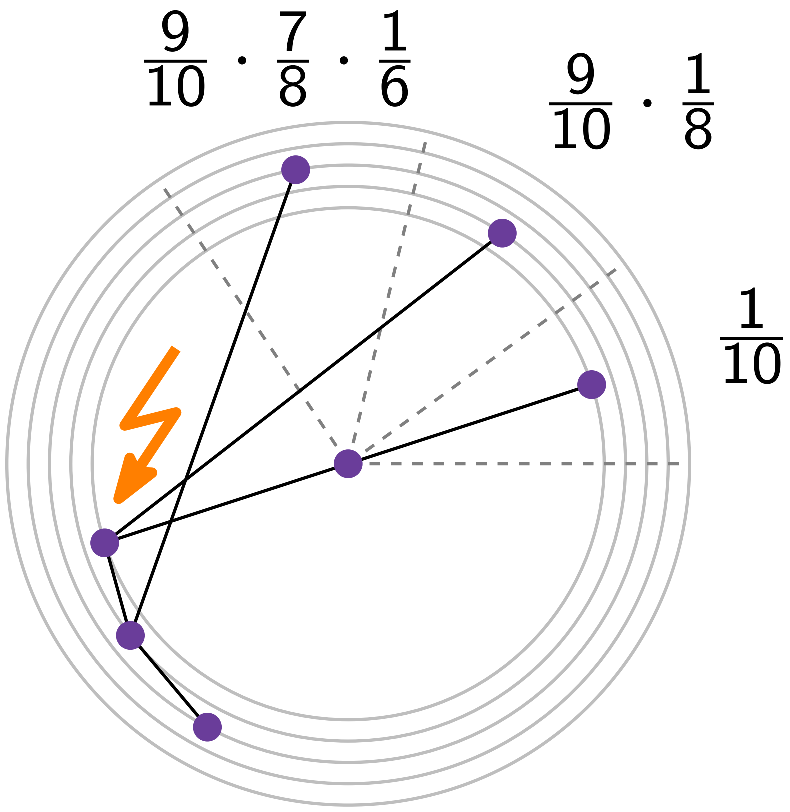
Idea

- Angle corresponding to size $\ell(u)$ of $T(u)$:

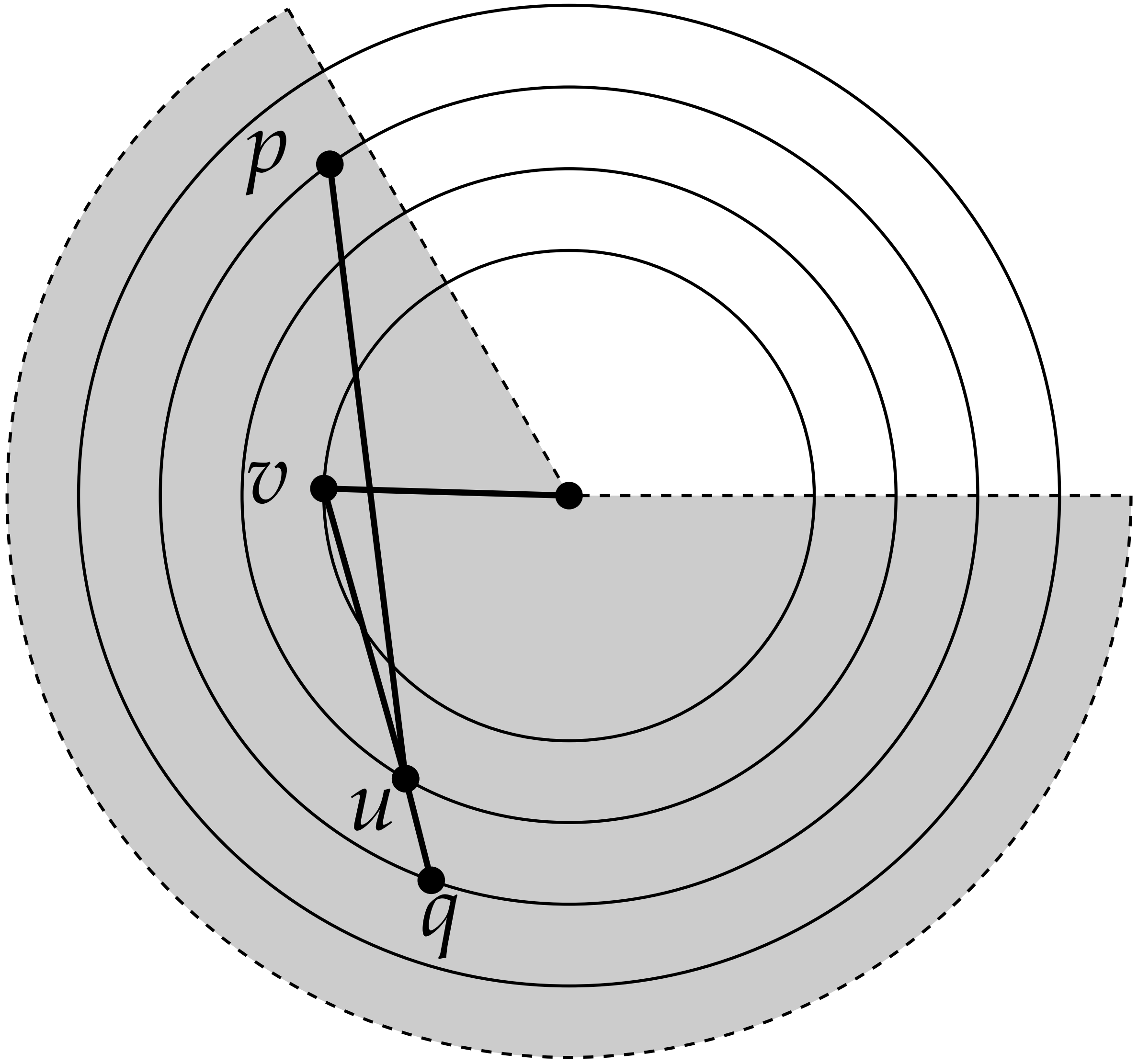


$$\tau_u = \frac{\ell(u)}{\ell(v) - 1}\tau_v$$

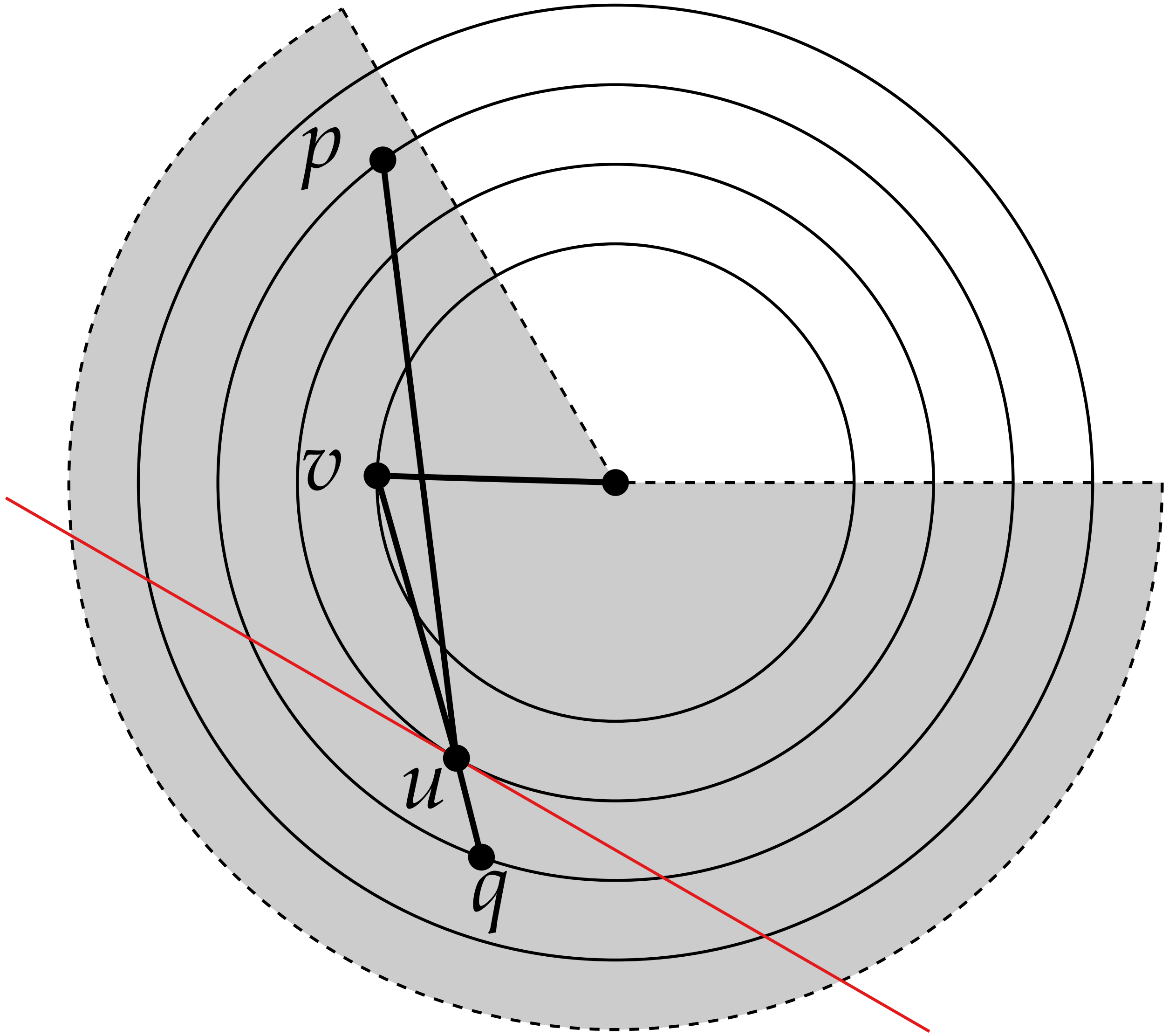




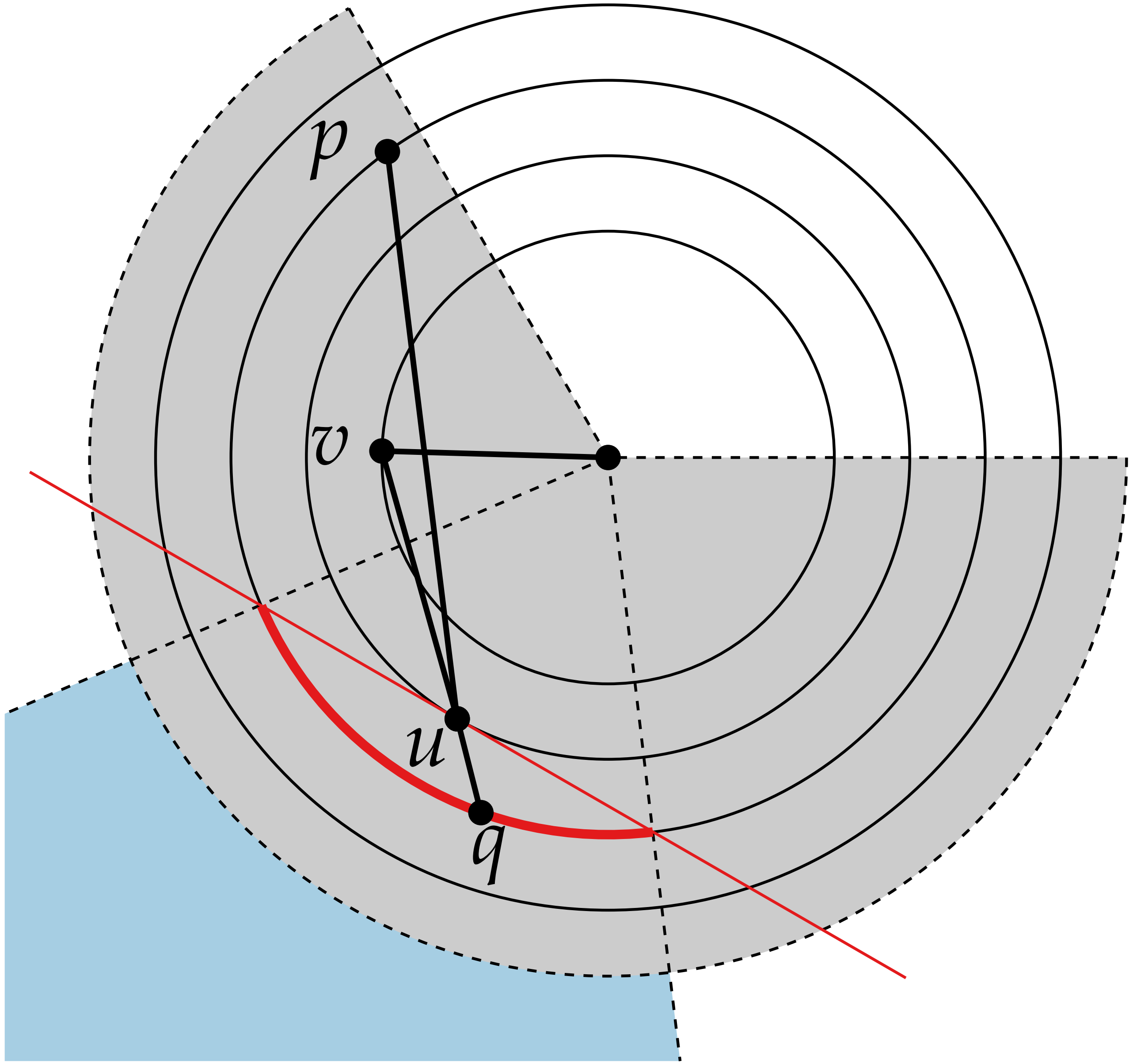
Radial layout – how to avoid crossings



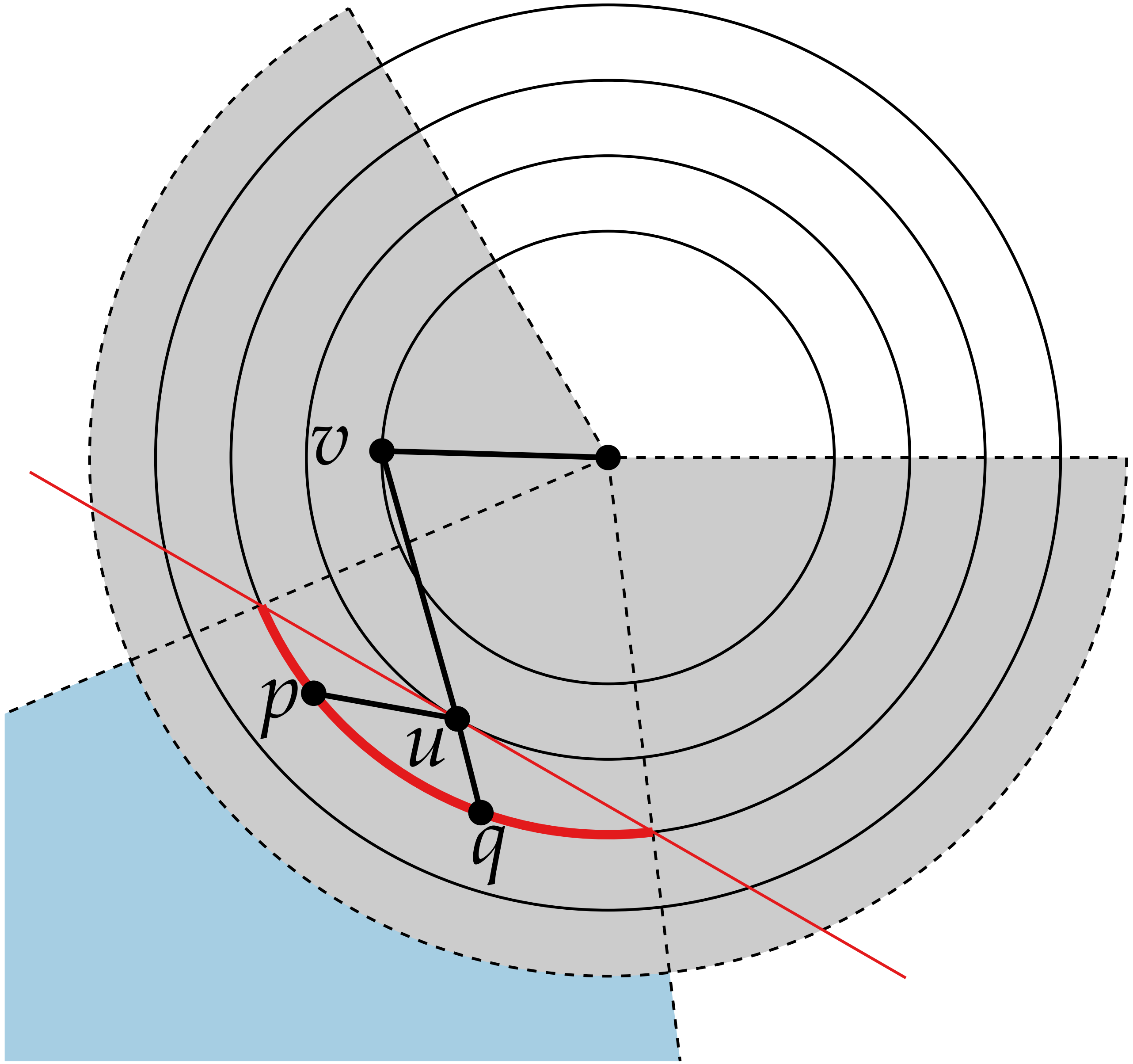
Radial layout – how to avoid crossings



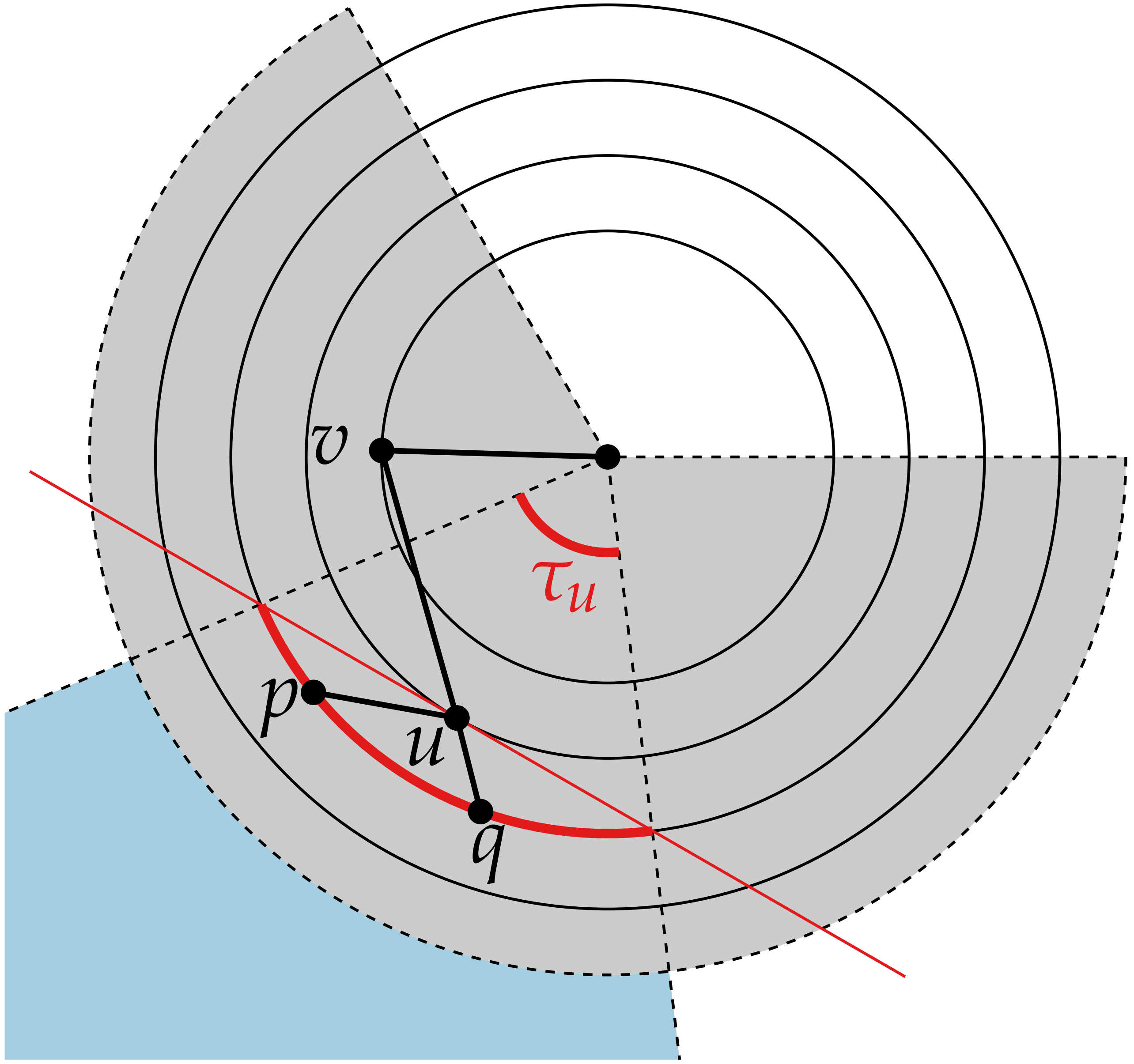
Radial layout – how to avoid crossings



Radial layout – how to avoid crossings

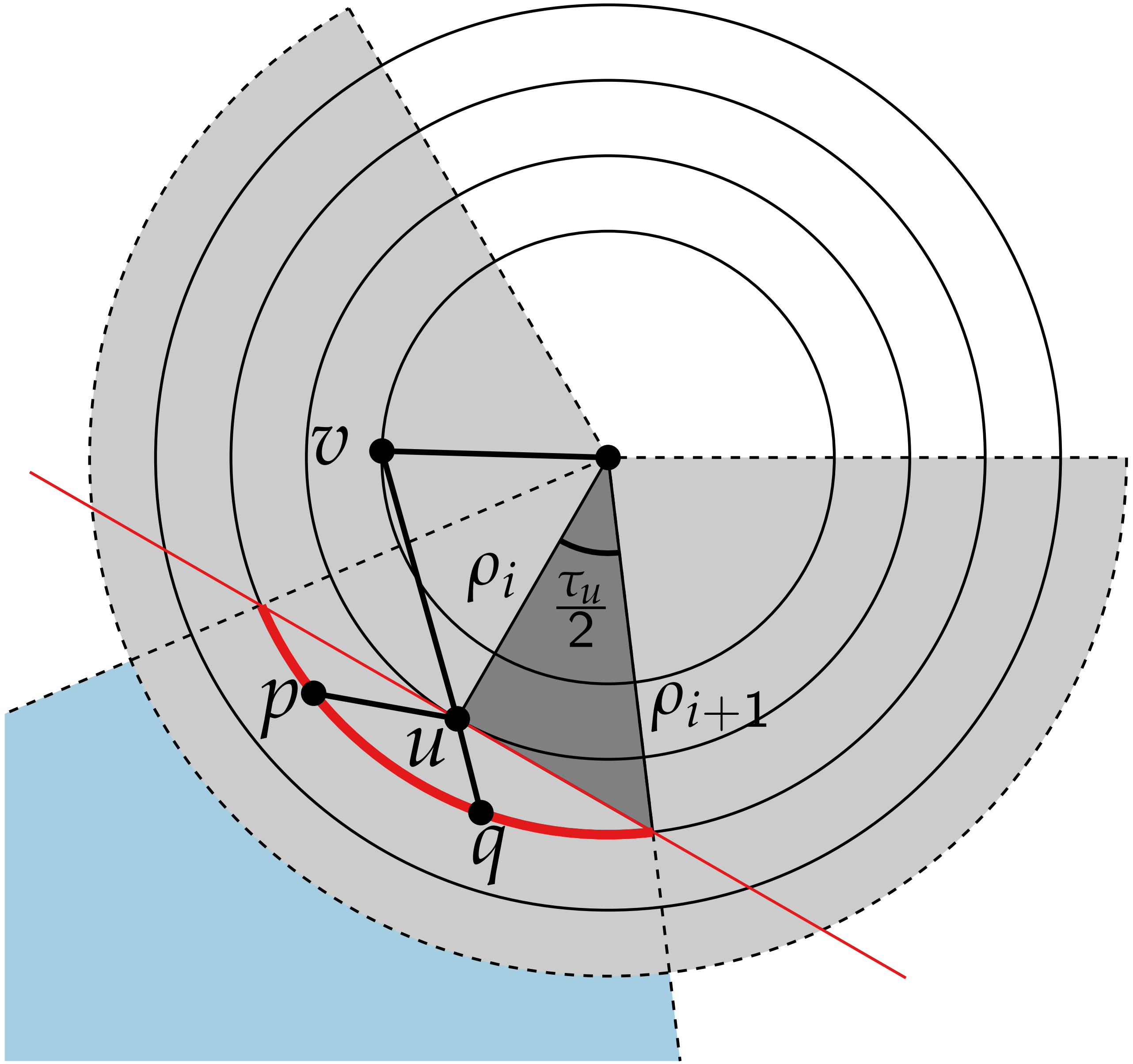


Radial layout – how to avoid crossings



- τ_u – angle of the wedge corresponding to vertex u

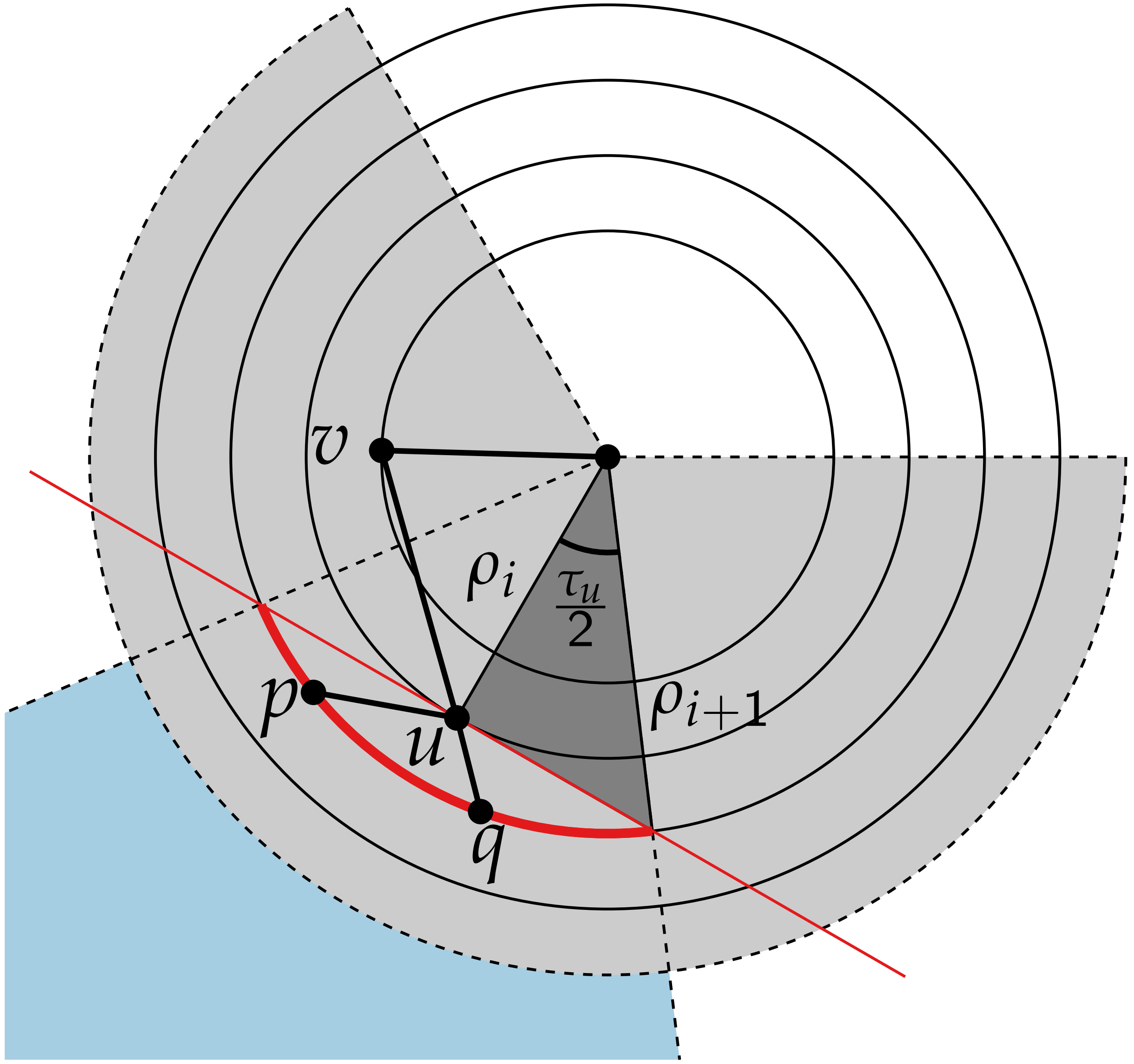
Radial layout – how to avoid crossings



- τ_u – angle of the wedge corresponding to vertex u
- $\ell(u)$ – number of nodes in the subtree rooted at u
- ρ_i – raduis of layer i

- $\cos \frac{\tau_u}{2} = \frac{\rho_i}{\rho_{i+1}}$

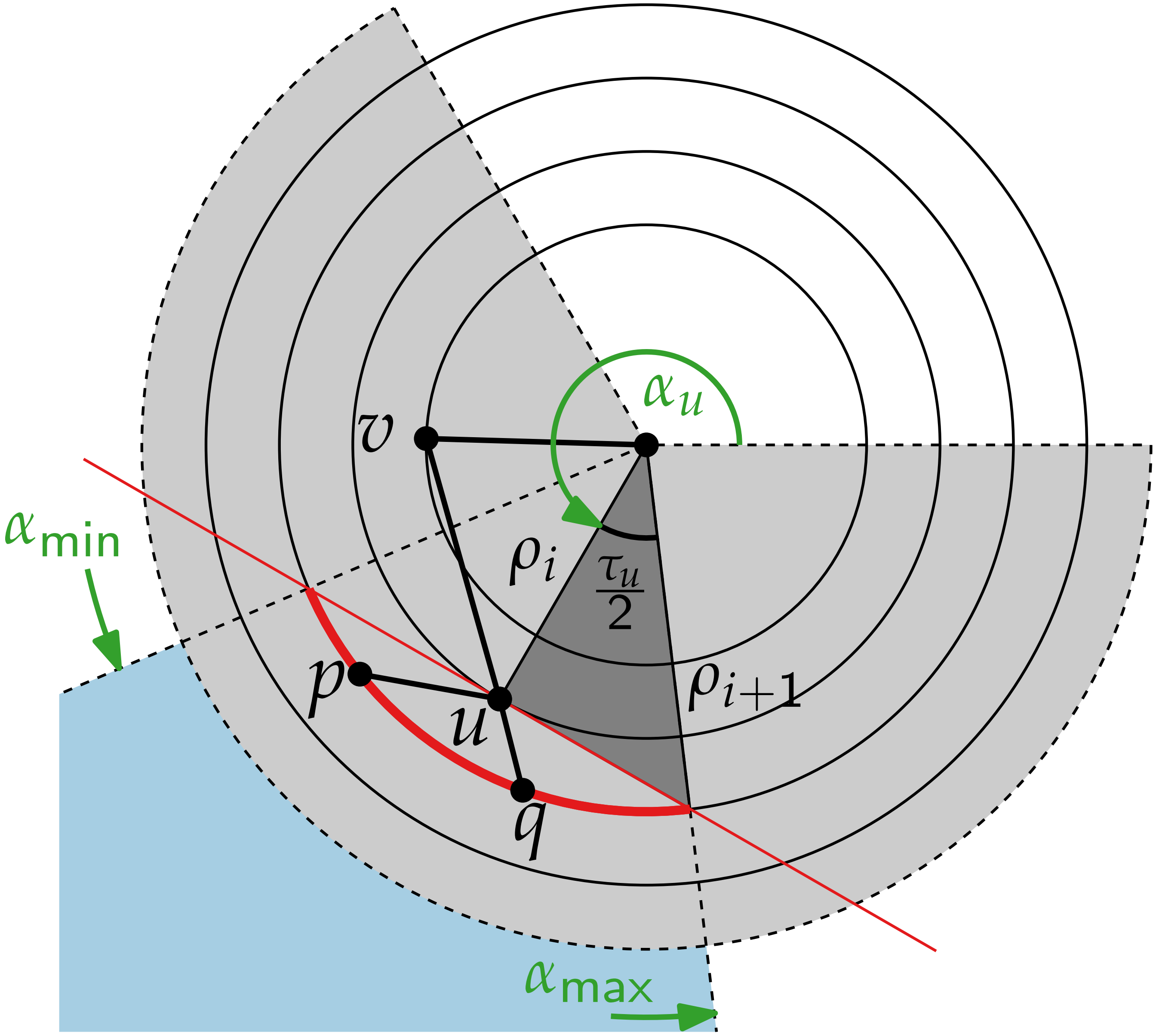
Radial layout – how to avoid crossings



- τ_u – angle of the wedge corresponding to vertex u
- $\ell(u)$ – number of nodes in the subtree rooted at u
- ρ_i – raduis of layer i

- $\cos \frac{\tau_u}{2} = \frac{\rho_i}{\rho_{i+1}}$
- $\tau_u = \min\{\frac{\ell(u)}{\ell(v)-1}\tau_v, 2\arccos \frac{\rho_i}{\rho_{i+1}}\}$

Radial layout – how to avoid crossings



- τ_u – angle of the wedge corresponding to vertex u
- $\ell(u)$ – number of nodes in the subtree rooted at u
- ρ_i – raduis of layer i

- $\cos \frac{\tau_u}{2} = \frac{\rho_i}{\rho_{i+1}}$
- $\tau_u = \min\{\frac{\ell(u)}{\ell(v)-1}\tau_v, 2 \arccos \frac{\rho_i}{\rho_{i+1}}\}$
- Alternative:
  $\alpha_{\min} = \alpha_u - \frac{\tau_u}{2} \geq \alpha_u - \arccos \frac{\rho_i}{\rho_{i+1}}$
  $\alpha_{\max} = \alpha_u + \frac{\tau_u}{2} \leq \alpha_u + \arccos \frac{\rho_i}{\rho_{i+1}}$

Radial layout – pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

begin
  $postorder(r)$
  $preorder(r, 0, 0, 2\pi)$
  return $(d_v, \alpha_v)_{v \in V(T)}$
  // vertex pos./polar coord.

postorder(vertex v)
  calculate the size of the subtree recursively

Radial layout – pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

begin
- $\mathit{postorder}(r)$
- $\mathit{preorder}(r, 0, 0, 2\pi)$
- **return** $(d_v, \alpha_v)_{v \in V(T)}$
- // vertex pos./polar coord.

postorder(vertex v)
- $\ell(v) \leftarrow 1$
- **foreach** child w of v **do**
  - $\mathit{postorder}(w)$
  - $\ell(v) \leftarrow \ell(v) + \ell(w)$

Radial layout – pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

begin
  postorder(r)
  preorder$(r, 0, 0, 2\pi)$
  return $(d_v, \alpha_v)_{v \in V(T)}$
  // vertex pos./polar coord.

postorder(vertex v)
  $\ell(v) \leftarrow 1$
  foreach child w of v **do**
    postorder(w)
    $\ell(v) \leftarrow \ell(v) + \ell(w)$

preorder(vertex v, t, $\alpha_{\min}$, $\alpha_{\max}$)
  $d_v \leftarrow \rho_t$
  $\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2$
  if $t > 0$ **then**
    $\alpha_{\min} \leftarrow \max\{\alpha_{\min}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}$
    $\alpha_{\max} \leftarrow \min\{\alpha_{\max}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}$
  $\mathit{left} \leftarrow \alpha_{\min}$
  foreach child w of v **do**
    $\mathit{right} \leftarrow \mathit{left} + \frac{\ell(w)}{\ell(v)-1} \cdot (\alpha_{\max} - \alpha_{\min})$
    preorder$(w, t+1, \mathit{left}, \mathit{right})$
    $\mathit{left} \leftarrow \mathit{right}$

Radial layout – pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \dots < \rho_k$)

```
begin
    postorder(r)
    preorder(r, 0, 0, 2π)
    return (d_v, α_v)_{v∈V(T)}
    // vertex pos./polar coord.
```

```
postorder(vertex v)
    ℓ(v) ← 1
    foreach child w of v do
        postorder(w)
        ℓ(v) ← ℓ(v) + ℓ(w)
```

```
preorder(vertex v, t, α_min, α_max)
    d_v ← ρ_t
    α_v ← (α_min + α_max)/2
    if t > 0 then
        α_min ← max{α_min, α_v − arccos ρ_t/ρ_{t+1}}
        α_max ← min{α_max, α_v + arccos ρ_t/ρ_{t+1}}
    left ← α_min
    foreach child w of v do
        right ← left + ℓ(w)/(ℓ(v)−1) · (α_max − α_min)
        preorder(w, t + 1, left, right)
        left ← right
```

Radial layout – pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

```
begin
    postorder(r)
    preorder(r, 0, 0, 2π)
    return (d_v, α_v)_{v∈V(T)}
    // vertex pos./polar coord.
```

```
postorder(vertex v)
    ℓ(v) ← 1
    foreach child w of v do
        postorder(w)
        ℓ(v) ← ℓ(v) + ℓ(w)
```

```
preorder(vertex v, t, α_min, α_max)
    d_v ← ρ_t                                   //output
    α_v ← (α_min + α_max)/2
    if t > 0 then
        α_min ← max{α_min, α_v − arccos ρ_t/ρ_{t+1}}
        α_max ← min{α_max, α_v + arccos ρ_t/ρ_{t+1}}
    left ← α_min
    foreach child w of v do
        right ← left + ℓ(w)/(ℓ(v)−1) · (α_max − α_min)
        preorder(w, t + 1, left, right)
        left ← right
```

Radial layout – pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

begin
- *postorder*(r)
- *preorder*$(r, 0, 0, 2\pi)$
- **return** $(d_v, \alpha_v)_{v \in V(T)}$
- // vertex pos./polar coord.

postorder(vertex v)
- $\ell(v) \leftarrow 1$
- **foreach** child w of v **do**
  - *postorder*(w)
  - $\ell(v) \leftarrow \ell(v) + \ell(w)$

preorder(vertex v, t, $\alpha_{\min}$, $\alpha_{\max}$)
- $d_v \leftarrow \rho_t$
- $\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2$ //*output*
- **if** $t > 0$ **then**
  - $\alpha_{\min} \leftarrow \max\{\alpha_{\min}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}$
  - $\alpha_{\max} \leftarrow \min\{\alpha_{\max}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}$
- *left* $\leftarrow \alpha_{\min}$
- **foreach** child w of v **do**
  - *right* $\leftarrow$ *left* $+ \frac{\ell(w)}{\ell(v)-1} \cdot (\alpha_{\max} - \alpha_{\min})$
  - *preorder*$(w, t+1, \mathit{left}, \mathit{right})$
  - *left* $\leftarrow$ *right*

Runtime?

Radial layout – pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

```
begin
    postorder(r)
    preorder(r, 0, 0, 2π)
    return (d_v, α_v)_{v∈V(T)}
    // vertex pos./polar coord.
```

```
postorder(vertex v)
    ℓ(v) ← 1
    foreach child w of v do
        postorder(w)
        ℓ(v) ← ℓ(v) + ℓ(w)
```

Runtime? $\mathcal{O}(n)$

```
preorder(vertex v, t, α_min, α_max)
    d_v ← ρ_t                          //output
    α_v ← (α_min + α_max)/2
    if t > 0 then
        α_min ← max{α_min, α_v − arccos ρ_t/ρ_{t+1}}
        α_max ← min{α_max, α_v + arccos ρ_t/ρ_{t+1}}
    left ← α_min
    foreach child w of v do
        right ← left + ℓ(w)/(ℓ(v)−1) · (α_max − α_min)
        preorder(w, t + 1, left, right)
        left ← right
```

Radial layout – pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

begin
  postorder(r)
  preorder$(r, 0, 0, 2\pi)$
  return $(d_v, \alpha_v)_{v \in V(T)}$
  // vertex pos./polar coord.

postorder(vertex v)
  $\ell(v) \leftarrow 1$
  foreach child w of v **do**
    postorder(w)
    $\ell(v) \leftarrow \ell(v) + \ell(w)$

preorder(vertex v, t, $\alpha_{\min}$, $\alpha_{\max}$)
  $d_v \leftarrow \rho_t$
  $\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2$ *//output*
  if $t > 0$ **then**
    $\alpha_{\min} \leftarrow \max\{\alpha_{\min}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}$
    $\alpha_{\max} \leftarrow \min\{\alpha_{\max}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}$
  $\mathit{left} \leftarrow \alpha_{\min}$
  foreach child w of v **do**
    $\mathit{right} \leftarrow \mathit{left} + \frac{\ell(w)}{\ell(v)-1} \cdot (\alpha_{\max} - \alpha_{\min})$
    preorder$(w, t+1, \mathit{left}, \mathit{right})$
    $\mathit{left} \leftarrow \mathit{right}$

Runtime? $\mathcal{O}(n)$

Correctness?

Radial layout – pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

begin
- *postorder*(r)
- *preorder*$(r, 0, 0, 2\pi)$
- **return** $(d_v, \alpha_v)_{v \in V(T)}$
- // vertex pos./polar coord.

postorder(vertex v)
- $\ell(v) \leftarrow 1$
- **foreach** child w of v **do**
  - *postorder*(w)
  - $\ell(v) \leftarrow \ell(v) + \ell(w)$

preorder(vertex v, t, α_{min}, α_{max})
- $d_v \leftarrow \rho_t$
- $\alpha_v \leftarrow (\alpha_{\text{min}} + \alpha_{\text{max}})/2$ //*output*
- **if** $t > 0$ **then**
  - $\alpha_{\text{min}} \leftarrow \max\{\alpha_{\text{min}}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}$
  - $\alpha_{\text{max}} \leftarrow \min\{\alpha_{\text{max}}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}$
- *left* $\leftarrow \alpha_{\text{min}}$
- **foreach** child w of v **do**
  - *right* $\leftarrow$ *left* $+ \frac{\ell(w)}{\ell(v)-1} \cdot (\alpha_{\text{max}} - \alpha_{\text{min}})$
  - *preorder*$(w, t+1, \textit{left}, \textit{right})$
  - *left* $\leftarrow$ *right*

Runtime? $\mathcal{O}(n)$

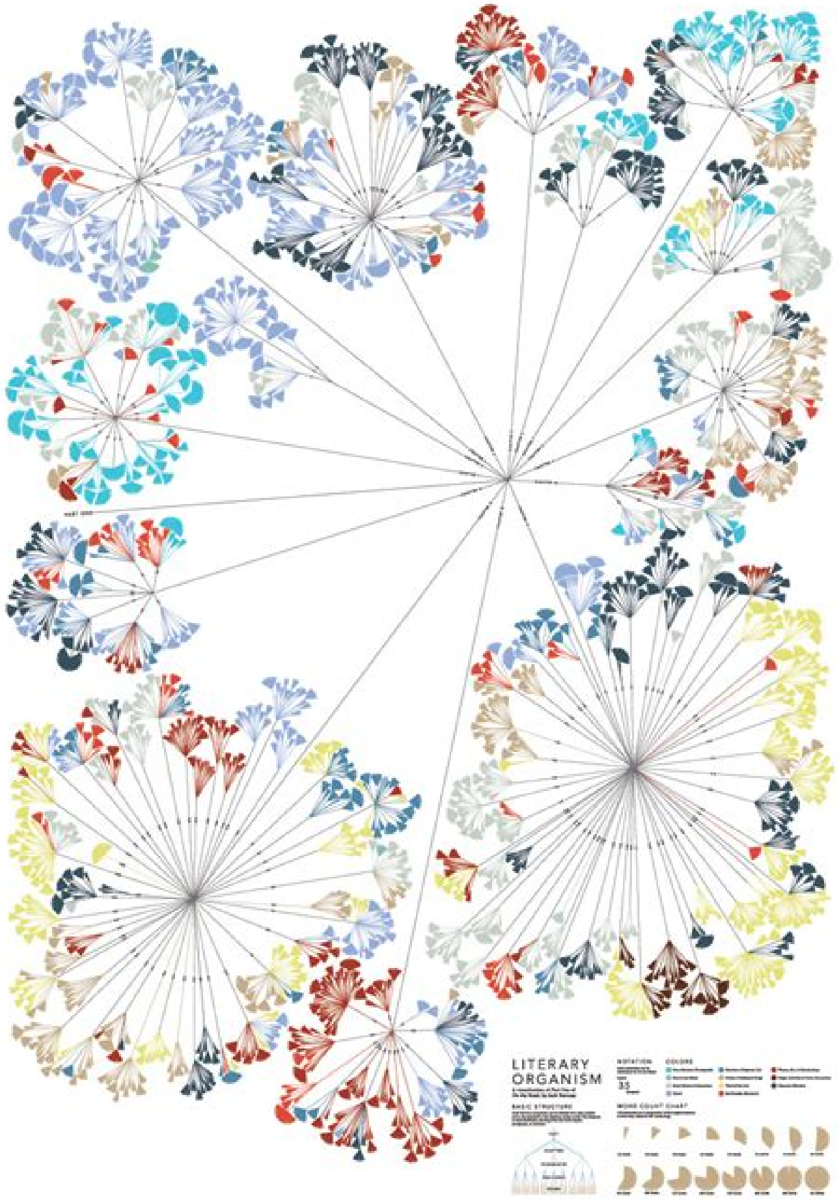
Correctness? ✓

Radial layout – result

Theorem.
Let T be a tree with n vertices. The RadialTreeLayout algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is radial drawing
- Vertices lie on circle according to their depth
- Area quadratic in max degree times height of T (see book if interested)

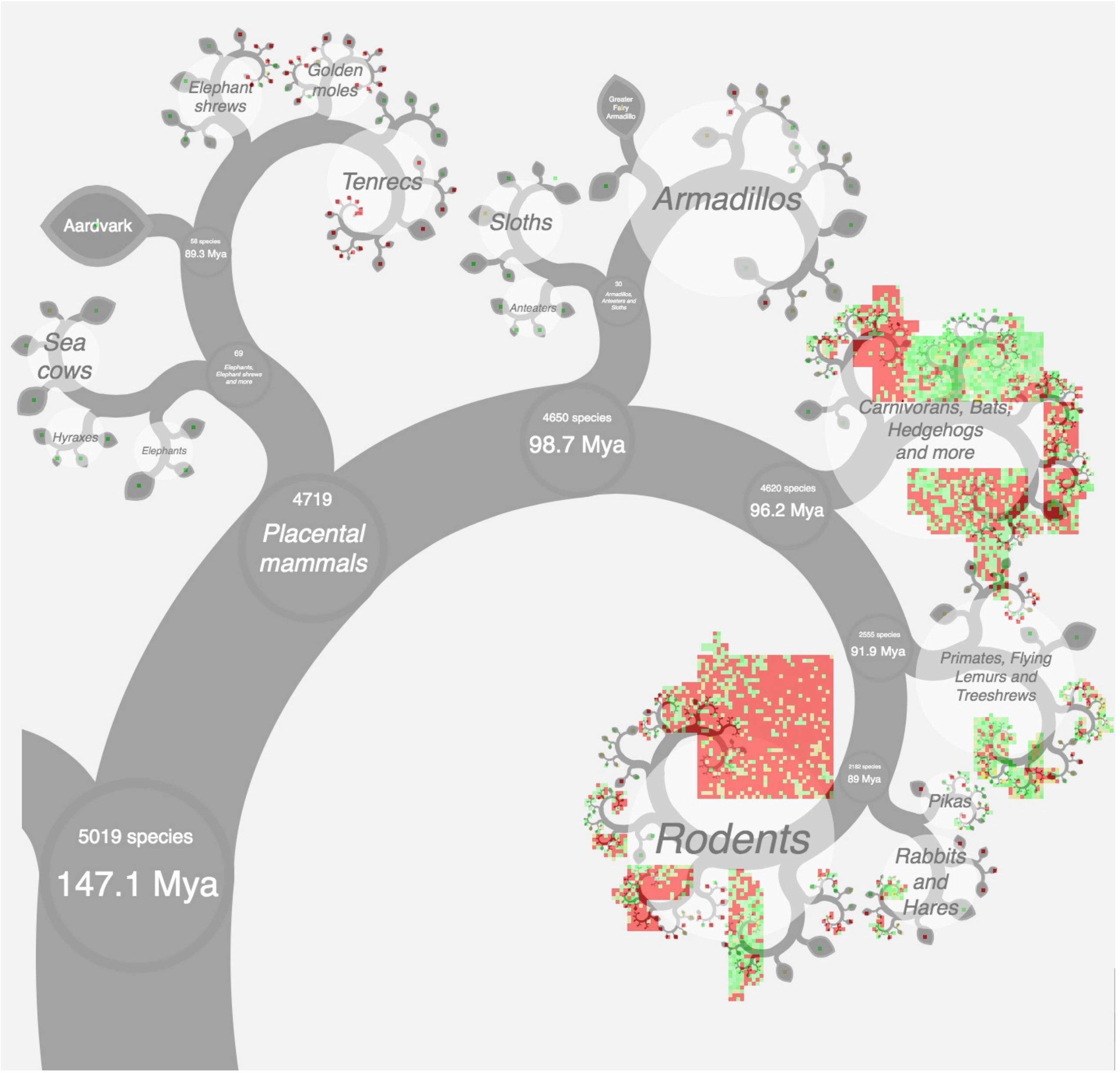
Other tree visualisation styles



Writing Without Words:
The project explores methods to visualises the differences in writing styles of different authors.

Similar to ballon layout

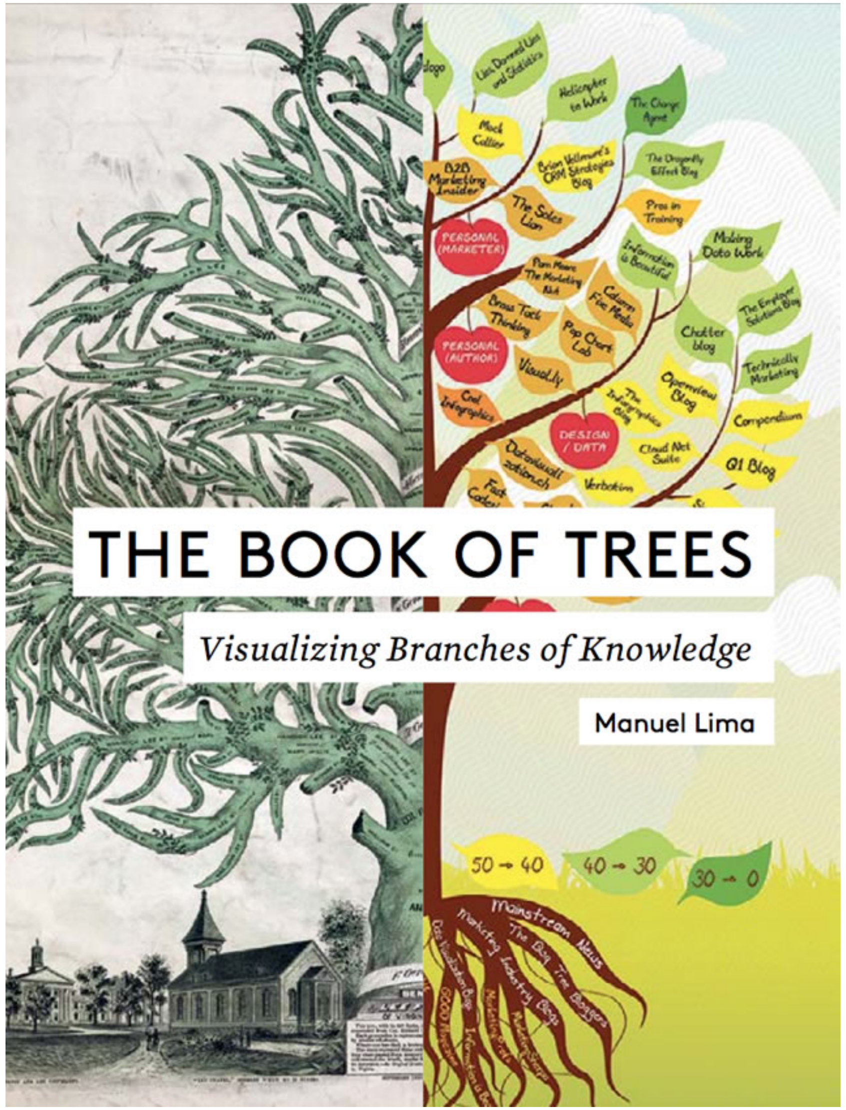
Other tree visualisation styles



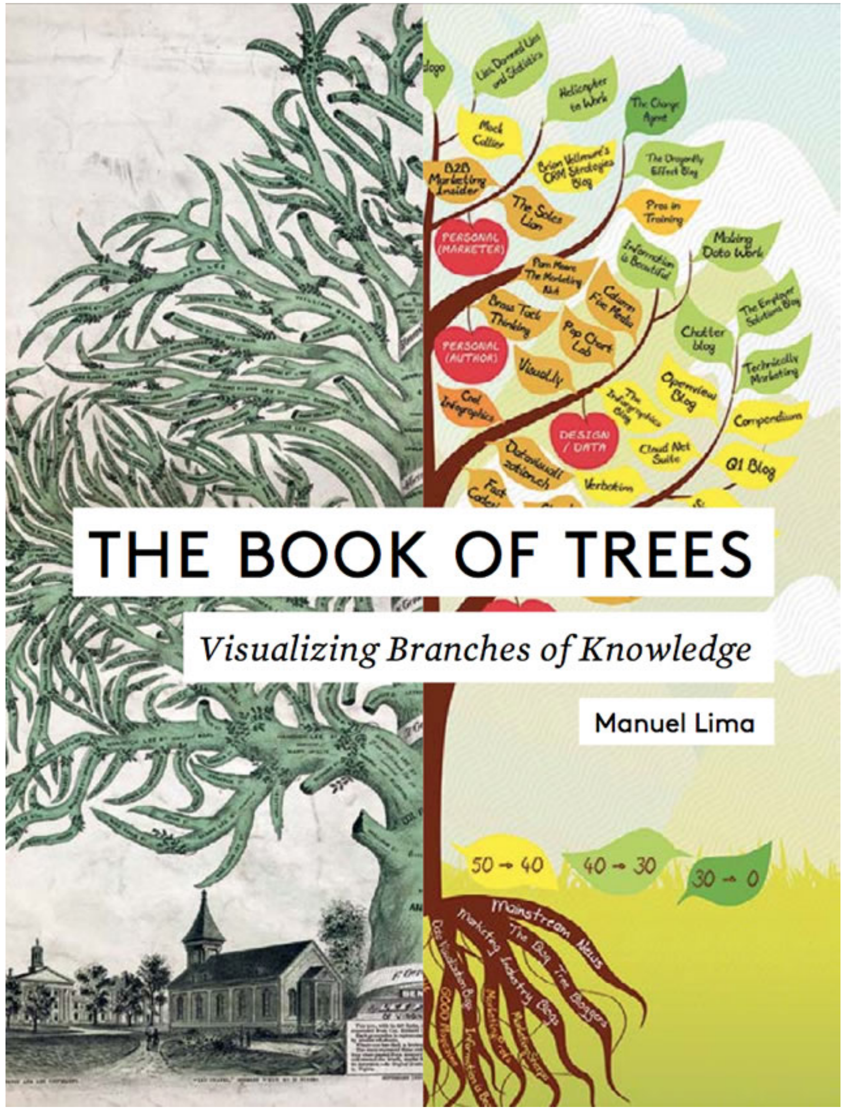
A phylogenetically organised display of data for all placental mammal species.

Fractal layout

Other tree visualisation styles



Other tree visualisation styles



treevis.net

Literature

- [GD Ch. 3.1] for divide and conquer methods for rooted trees
- [RT81] Reingold and Tilford, "Tidier Drawings of Trees" 1981 – original paper for level-based layout algo
- [SR83] Reingold and Supowit, "The complexity of drawing trees nicely" 1983 – NP-hardness proof for area minimisation & LP
- `treevis.net` – compendium of drawing methods for trees
  (links on website)
- [GD Ch. 3.2] for divide an conquer mehtods for series-parallel graphs